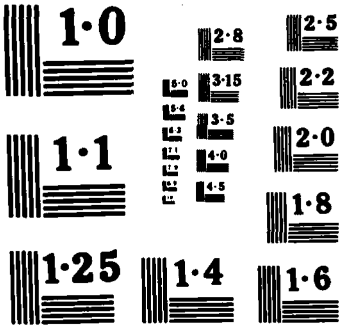
AD-A155 724 A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND 1/4
AND CONTROL APPLICATIONS PART 2(U) COMPUTER CORP OF
AMERICA CAMBRIDGE MA 30 JAN 80 CCA-80-04-PT-2
UNCLASSIFIED N00039-77-C-0074 F/G 9/2 NL



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

A Distributed Database Management System for Command and Control Applications: Final Technical Report—Part II

1

Technical Report
CCA-80-04
January 30, 1980

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

A Distributed Database Management System for Command and Control Applications
FINAL TECHNICAL REPORT - Part II

January 1, 1977 to January 31, 1980

This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract No. N00039-77-C-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

1. Introduction

This report contains a collection of papers that detail the design of a SDD-1 (A System for Distributed Databases). It is the second part of the final technical report for a research project called "A Distributed Database Management System for Command and Control Applications".

Each paper included in this report is a self-contained entity in that the papers can be read in any order without loss of understanding. However, if a reader is not specifcally interested in one particular apsect of SDD-1, it is suggested that the first paper be read first. The first paper presents the overall design of SDD-1. The second paper describes the SDD-1 concurrency control algorithm in detail. The third paper is a proof of the correctness of the SDD-1 concurrency control algorithm. The fourth paper describes the reliability mechanisms of SDD-1. The last paper focuses on the distributed query processing technique used by SDD-1.

SDD-1:
A System for Distributed Databases

(Revised)

J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman
M. Hammer, T. A. Landers, C. Reeve, D. Shipman, E. Wong

August 1, 1979

Computer Corporation of America
575 Technology Square
Cambridge,Massachusetts 02139

This research was supported by the Defense Advanced Research Project Agency of the Department of Defense under Contract No. N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the U. S. Government.

Abstract

The declining cost of computer hardware and the increasing data processing needs of geographically dispersed organizations has led to substantial interest in distributed data management. SDD-1 is a distributed database management system currently being developed by Computer Corporation of America. Users interact with SDD-1 precisely as if it were a non-distributed database system, because SDD-1 handles all issues arising from the distribution of data. These issues include distributed concurrency control, distributed query processing, resiliency to component failure, and distributed directory management. This paper presents an overview of the SDD-1 design and its solutions to the above problems.

This paper is the first of a series of companion papers on SDD-1 [BERNSTEIN and SHIPMAN] [BERNSTEIN et al. b] [GOODMAN et al.] [HAMMER and SHIPMAN].

Table of Contents

1. Introduction

SDD-1 is a distributed database management system under development by Computer Corporation of America. SDD-1 is a system for managing databases whose storage is distributed over a network of computers. Functionally, SDD-1 provides the same capabilities that one expects of any modern database management system (abbr. DBMS), and users interact with it precisely as if it were not distributed.

Systems like SDD-1 are appropriate for applications which exhibit two characteristics: First, the activity requires an integrated database. That is, the activity entails access to a single pool of information by multiple persons, organizations, or programs. And second, either the users of the information or its sources are distributed geographically. Many data processing applications have these characteristics, including:

- inventory control, accounting, personnel information, and similar data processing systems in large companies;

- point-of-sale accounting systems, electronic banking, and other consumer oriented on-line processing systems;
- large-scale data resources, e.g., census, climatology, toxicology and similar databases;
- military intelligence databases, and command and control systems;
- report generating systems for businesses with multiple data processing centers; and so forth.

Decentralized processing is desirable in these applications for reasons of performance, reliability, and flexibility of function. Centralized control is needed to ensure operation in accordance with overall policy and goals. By meeting both these goals in one system, distributed database management offers unique benefits.

However, distributed database systems pose new technical challenges due to their inherent requirements for data communication and their inherent potential for parallel processing. The principal bottleneck in these systems is data communication. All economically feasible long distance communication media incur lengthy delays and/or low bandwidth. Moreover, the cost of moving data through a network is comparable to the cost of storing it locally for many days. Parallel processing is also an inherent aspect of distributed systems and mitigates to some extent the communication factor. However, it is often difficult to construct algorithms that can exploit parallelism.

For these reasons, the techniques used to implement centralized DBMSs must be re-examined in the distributed DBMS context. We have done this in developing SDD-1 and this paper surveys our main results.

Section 2 describes SDD-1's overall architecture and the flow of events in processing transactions. Sections 3 - 5 then introduce the techniques used by SDD-1 for solving the most difficult problems in distributed data management: concurrency control, query processing, and reliability. Detailed discussions of these techniques are presented in [BERNSTEIN et al. a,b], [BERNSTEIN and SHIPMAN], [GOODMAN et al.], [HAMMER and SHIPMAN], and [WONG]. Section 6 explains how these techniques are used to handle the management of system directories. The paper concludes with a brief history of SDD-1 and a summary of its principal contributions to the field.

2. System Organization

2.1 Data Model

SDD-1 supports a relational data model [CODD]. Users interact with SDD-1 in a high-level language called Datalanguage [CCA] which is illustrated in Figure 2.1. Datalanguage differs from relational languages such as QUEL [HELD et al.] or SEQUEL [CHAMBERLIN et al.] primarily in its use of "declared" variables. This construct and related control structures expand the power of Datalanguage to that of a general purpose programming language. For purposes of this paper, the differences between Datalanguage and QUEL or SEQUEL are not important, and for pedagogic ease, we adopt QUEL terminology.

Datalanguage may be used as a query language for end-users, but is more typically invoked by host programs. Datalanguage is embedded in host programs in essentially the same manner as QUEL or SEQUEL. That is, the host program issues self-contained Datalanguage commands to

A Datalanguage Command Figure 2.1

```
Relation:  CUSTOMER(Name,Branch,Acct#,SavBal,ChkBal,LoanBal)

Command:   Update C in CUSTOMER with C.Name = "Adams"
              Begin
                 C.SavBal = C.SavBal - 100
                 C.ChkBal = C.ChkBal + 100
              End;
```

SDD-1, which processes these commands exactly as if entered by an end-user.

A single Datalanguage command is called a transaction (e.g., the command shown in Figure 2.1 is a transaction). Transactions are the units of atomic interaction between SDD-1 and the external world. This concept of transaction is similar to that of INGRES [HELD et al.] and System R [ASTRAHAN et al].

An SDD-1 database consists of (logical) relations. Each SDD-1 relation is partitioned into sub-relations called logical fragments which are the units of data distribution. Logical fragments are defined in two steps. First, the relation is partitioned horizontally into subsets defined by "simple" restrictions.* Then each

*A simple restriction is a boolean expression whose clauses are of the form <attribute> <rel_op> <constant>, where <rel_op> is =, ≠, >, <, etc.

horizontal subset is partitioned into subrelations defined by projections. (See Figures 2.2, 2.3.) To reconstruct the logical relation from its fragments, a unique tuple identifier is appended to each tuple and included in every fragment [ROTHNIE and GOODMAN], [DAYAL and BERNSTEIN].

Logical fragments are the unit of data distribution, meaning that each may be stored at any one or several sites in the system. The definition of logical fragments and the assignment of fragments to sites occurs when the database is designed. A stored copy of a logical fragment is called a stored fragment.

Note that user transactions are unaware of data distribution or redundancy. They reference only relations, not fragments. It is SDD-1's responsibility to translate from relations to logical fragments, and then to select the stored fragments to access in processing any given transaction.

Horizontal Partitioning Figure 2.2

CUSTOMER	(Name,	Branch,	Acct#,	SavBal,	ChkBal,	LoanBal)
CUST_1	Wash. . .	1	1234	$100	$200	-$8
CUST_2	Jeff.	2	5678	$200	$300	$30000
CUST_3a	Adams	3	9012	$1000	$0	$20000
CUST_3b	Munroe	3	3456	$100	$50	$0

CUST_1 = CUST where Branch = 1
CUST_2 = CUST where Branch = 2
CUST_3a = CUST where Branch = 3 and LoanBal ≠0
CUST_3b = CUST where Branch = 3 and LoanBal = 0

Vertical Partitioning Figure 2.2

<table>
<tr><th>CUSTOMER</th><th>(Name,</th><th>Branch,</th><th>Acct#,</th><th>SavBal,</th><th>ChkBal,</th><th>LoanBal)</th></tr>
<tr><td>CUST. 1</td><td colspan="2">CUST_1.1</td><td colspan="4">CUST_1.2</td></tr>
<tr><td>CUST_2</td><td colspan="3">CUST_2.1</td><td colspan="3">CUST_2.2</td></tr>
<tr><td>CUST_3a</td><td colspan="2">CUST_3a.1</td><td colspan="2">CUST_3a.2</td><td colspan="2">CUST_3a.3</td></tr>
<tr><td>CUST_3b</td><td colspan="2">CUST_3b.1</td><td colspan="2">CUST_3b.2</td><td>CUST_3b.3</td><td>CUST_3b.4</td></tr>
</table>

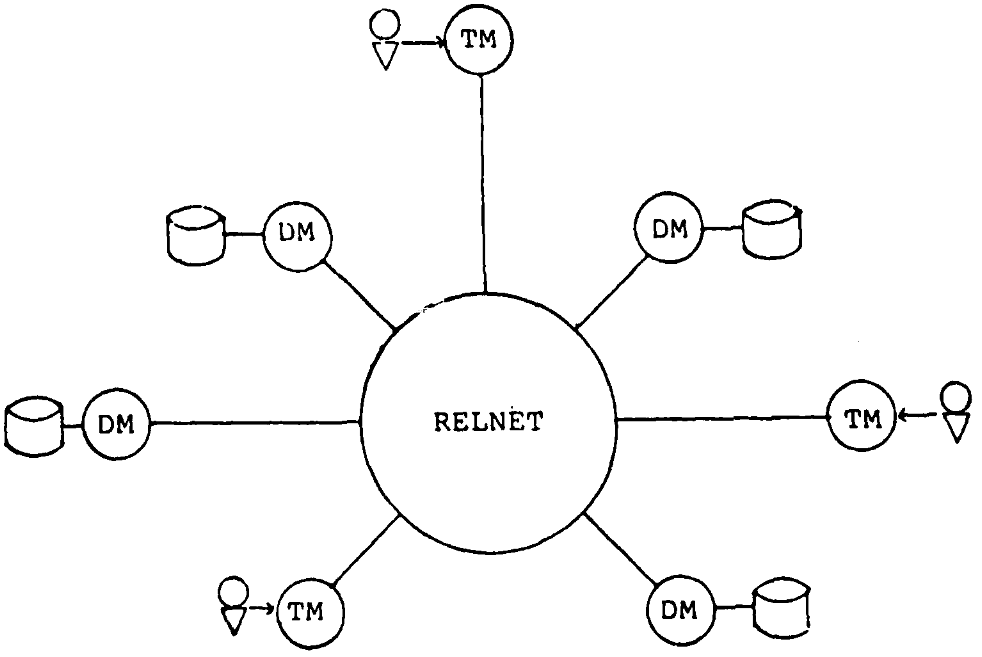
CUST_1.1 = CUST_1 [Name, Branch]
CUST_1.2 = CUST_1 [Acct#, SavBal, ChkBal, LoanBal]
etc.

In order to reconstruct CUSTOMER from its fragments, a unique tuple identifier is appended to each tuple and included in every fragment [ROTHNIE and GOODMAN].

2.2 General Architecture

SDD-1 is a collection of three types of <u>virtual machines</u> [HORNING and RANDELL] -- Transaction Modules (TMs), Data Modules (DMs), and a Reliable Network (RelNet) -- configured as in Figure 2.4.

SDD-1 Configuration Figure 2.4



All data managed by SDD-1 is stored by Data Modules (DMs). DMs are, in effect, back-end DBMSs that respond to commands from Transaction Modules. DMs respond to four types of commands: (1) Read part of the DM's database into a local workspace at that DM; (2) Move part of a local workspace from this DM to another DM; (3) Manipulate data in a local workspace at the DM; (4) Write part of the local workspace into the permanent database stored at the DM.

Transaction Modules (TMs) plan and control the distributed execution of transactions. Each transaction processed by SDD-1 is supervised by some TM, which performs these tasks: (1) Fragmentation -- it translates queries on relations into queries on logical fragments and decides which instances of stored fragments to access. (2) Concurrency control -- the TM synchronizes the transaction with all other active transactions in the system. (3) Access planning -- the TM compiles the transaction into a parallel program which can be executed cooperatively by several DMs. (4) Distributed query execution -- the TM coordinates execution of the compiled access plan, exploiting parallelism whenever possible.

The third SDD-1 virtual machine is the Reliable Network (RelNet) which interconnects TMs and DMs in a robust fashion. The RelNet provides four services: (1) Guaranteed delivery, allowing messages to be delivered even if the recipient is down at the time the message is sent, and even if the sender and receiver are never up simultaneously. (2) Transaction control, a mechanism for posting updates at multiple DMs, guaranteeing that either all DMs post the update or none do. (3) Site monitoring, to keep track of which sites have failed, and to inform sites impacted by failures. (4) Network clock, a virtual clock kept approximately synchronized at all sites.

This architecture divides the distributed DBMS problem into three pieces: database management, management of distributed transactions, and distributed DBMS reliability. By implementing each of these pieces as a self-contained virtual machine, the overall SDD-1 design is substantially simplified.

2.3 Run-Time Structure

Among the functions required to execute a transaction in a distributed DBMS, three are especially difficult: concurrency control, distributed query processing, and reliable posting of updates. SDD-1 handles each of these problems in a distinct processing phase, so that each can be solved independently. Consider transaction T of Figure 2.1. When T is submitted to SDD-1 for processing, the system invokes a three phase processing procedure. The phases are called Read, Execute, and Write.

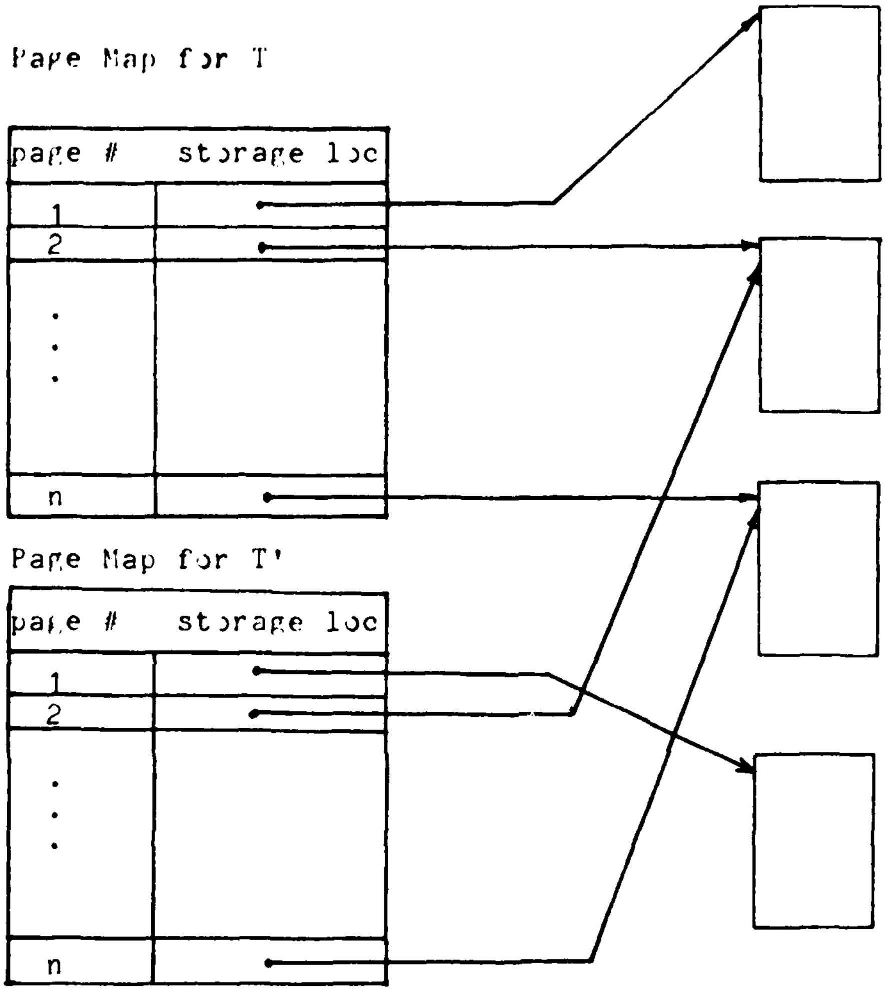
The first phase is the Read phase and exists for purposes of concurrency control. The TM that is supervising T analyzes it to determine which portions of the (logical) database it reads, called its read-set. In this case the TM would determine that T's read-set is

{C.SavBal, C.ChkBal | C.Name = "Adams"}.

In addition the TM decides which stored fragments to access to obtain that data. Then the TM issues Read commands to the DMs that house those fragments, instructing each DM to set aside a private copy of that fragment for use during subsequent processing phases.

The private copies obtained by the Read phase are guaranteed to be consistent even though the copies reside at distributed sites. The techniques for guaranteeing consistency are described in Section 3. Since the data is consistent when read, and since the copies are private, subsequent phases can operate freely on this data without fear of interference from other transactions.

We emphasize that no data is actually transferred between sites during the Read phase. Each DM simply sets aside the specified data in a workspace at the DM. Moreover, in each DM, the private workspace is implemented using a differential file mechanism [SEVERANCE and LOHMAN], so data is not actually copied. This mechanism operates as follows. The primary organization of a stored fragment is a paged file, much like a UNIX [RITCHIE and THOMPSON] or TENEX [BOBROW et al.] file. A page is a unit of logical storage; a page map is a function that associates a physical storage location with each page (see figure 2.5). The "private copy" set aside by a Read command is in reality a page map. Page maps behave like private copies because pages are never updated in place; if page P is modified on behalf of transaction T', say, a new block of secondary storage is allocated, and the modified page is written there. T' is able to access the modified page because its page map is also modified to reflect P's new version to storage mapping? Figure 2.5



T and T' share each page until one transaction or the other modifies the page.

storage location. Other transactions are unaffected because their page maps remain unchanged. A similar mechanism is described in [LORIE].

The second phase, called the Execute phase, implements distributed query processing. At this time, the TM compiles T into a distributed program that takes as input the distributed workspace created by the Read phase. This compilation procedure is described in Section 4. The compiled program consists of Move and Manipulate commands which cause the DMs to perform the intent of T in a distributed fashion. The compiled program is supervised by the TM to ensure that commands are sent to DMs in the correct order and to handle run-time errors.

The output of the program is a list of data items to be written into the database (in the case of update transactions) or displayed to the user (in the case of retrievals). In our example, this output list would contain a unique tuple identifier for Adams' tuple, identifiers for the field names SavBal and ChkBal, and the new values for these fields. This output list is produced in a workspace (i.e., temporary file) at one DM, and is not yet installed into the permanent database. Consequently, problems of concurrency control and reliable writing are irrelevant during this phase.

The final phase, called the Write phase, installs data modified by T into the permanent database and/or displays data retrieved by T to the user. For each entry in the output list, the TM determines which DM(s) contain copies of that data item. The TM orders the final DM that holds the output list to send the appropriate entries of the output list to each DM; it then issues Write commands to each of these DMs thereby causing the new values to be installed into the database. Techniques described in Section 5 are used during the Write phase to ensure that partial results are not installed or displayed even if multiple sites or communication links fail in mid-stream. This is the most difficult aspect of distributed DBMS reliability, and by separating it into a distinct phase, we simplify both it and the other phases.

The three-phase processing of transactions in SDD-1 neatly partitions the key technical problems of distributed database management. The next sections of this paper explain how SDD-1 solves each of these independent problems.

3. Concurrency Control

The problems that arise when multiple users access a shared database are well-known. Generically there are two types of problems: (1) If transaction T1 is reading a portion of the database while transaction T2 is updating it, T1 might read inconsistent data (see Figure 3.1). (2) If transactions T3 and T4 are both updating the database, race conditions can produce erroneous results (see Figure 3.2). These problems arise in all shared databases -- centralized or distributed -- and are conventionally solved using database locking. However, we have developed a new technique for SDD-1.

Reading Inconsistent Data Figure 3.1

Consider the database of Figures 2.2 and 2.3, and assume fragments CUST_3a.2, CUST_3a.3, are stored at different DMs:

```
Let transaction T1 be
   Range of C is CUSTOMER;
   Retrieve C (SavBal+ChkBal) where C.Name="Adams";
Let transaction T2 be
   Range of C is CUSTOMER;
   Replace C (SavBal=SavBal-$100, ChkBal=ChkBal+$100)
              where C.Name="Adams";
```

And suppose T1 & T2 execute in the following concurrent order

```
   T1  reads Adam's SavBal (=$1000) from fragment CUST_3a.2
   T2 writes Adam's SavBal (= $900) into fragment CUST_3a.2
   T2 writes Adam's ChkBal (= $100) into fragment CUST_3a.3
   T1  reads Adam's ChkBal (= $100) from fragment CUST_3a.3
```

T1's output will be $1000+$100=$1100, which is incorrect.

Race Condition Producing Erroneous Update Figure 3.2

Given the database of Figures 2.2 and 2.3.

```
Let transaction T3 be
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal+$100) where C.Name="Munroe";
Let transaction T4 be
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal- $50) where C.Name="Munroe";
```

And suppose T3 and T4 execute in the following concurrent order

```
   T3  reads Munroe's ChkBal   (=$50)
   T4  reads Munroe's ChkBal   (=$50)
   T4 writes Munroe's ChkBal   (=$0 )
   T3 writes Munroe's ChkBal   (=$50 + $100= $150)
```

The value of ChkBal left in the database is $150, which is incorrect. The final balance should be $50-$50+$100=$100.

3.1 Methodology

SDD-1, like most other DBMSs, adopts serializability as its criterion for concurrent correctness. Serializability requires that whenever transactions execute concurrently, their effect must be identical to some serial (i.e., non-interleaved) execution of those same transactions. This criterion is based on the assumption that each transaction maps a consistent database state into another consistent state. Given this assumption, every serial execution preserves consistency. Since a serializable execution is equivalent to a serial one, it too preserves database consistency.

Most DBMSs ensure serializability through database locking. By locking, we mean a synchronization method in which transactions dynamically reserve data before accessing it [ESWARAN et al].

SDD-1 uses two synchronization mechanisms that are distinctly different from locking [BERNSTEIN et al. c]. The first mechanism, called conflict graph analysis, is a technique for analyzing "classes" of transactions to detect those transactions that require little or no synchronization. The second mechanism consists of a set of *synchronization protocols* based on "timestamps", which synchronize those transactions that need it.

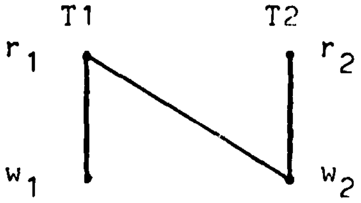
3.2 Conflict Graph Analysis

The *read-set* of a transaction is the portion of the database it reads and its *write-set* is the portion of the database it updates. Two transactions *conflict* if the read-set or write-set of one intersects the write-set of the other. In a system that uses locking, each transaction locks data before accessing it, so conflicting transactions never run concurrently. However, not all conflicts violate serializability; that is, some conflicting transactions can safely be run concurrently. More concurrency can be attained by checking whether or not a given conflict is troublesome, and only synchronizing those that are. Conflict graph analysis is a technique for doing this.

The nodes of a conflict graph represent the read-sets and write-sets of transactions, and edges represent conflicts among these sets. (There is also an edge between the read-set and write-set of each transaction.) Figure 3.3 shows sample conflict graphs. The important property is

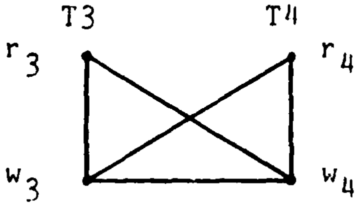
Conflict Graphs   Figure 3.3

```
Define transactions T1 and T2 as in Figure 3.1
   read-set (T1) = {C.SavBal, C.ChkBal s.t. C.Name="Adams"}
   write-set(T1) = {}
   read-set (T2) = read-set(T1)
   write-set(T2) = read-set(T1)
```



Note: nodes r_i and w_i denote the read-set and write-set of transaction T_i

```
Define transaction T3 and T4 as in Figure 3.2
   read-set (T3) = {C.ChkBal s.t. C.Name="Munroe"}
   write-set(T3) = read-set(T3)
   read-set (T4) = read-set(T3)
   write-set(T4) = read-set(T3)
```



that different kinds of edges require different levels of synchronization, and that synchronization as strong as locking is required only for edges that participate in cycles [BERNSTEIN and SHIPMAN]. In Figure 3.3, for example, transactions T1 and T2 do not require synchronization as strong as locking, whereas T3 and T4 do.

It is impractical to use conflict graph analysis at run-time because too much inter-site communication would be required to exchange information about conflicts. Instead we apply the technique off-line, during database design, as follows: the database administrator defines transaction classes, which are named groups of commonly executed transactions. Each class is defined by its name, a read-set, a write-set, and the TM at which it runs; a transaction is a member of a class if the transaction's read-set and write-set are contained in the class's read-set and write-set respectively. Conflict graph analysis is actually performed on these transaction classes, not on individual transactions as in Figure 3.3. (Notice that transactions from different classes can conflict only if their classes conflict.) The output of the analysis is a table telling for each class: (a) which other classes it conflicts with, and (b) for each such conflict, how much synchronization (if any) is required to ensure serializability.

It is convenient to assume that each TM is only permitted to supervise transactions from one class, and vice versa.* At run-time, when transaction T is submitted, the system determines which class(es) T is a member of, and sends T

* This assumption engenders no loss of generality since several TMs can be multi-programmed at one site, and several classes can be defined with identical read-sets and write-sets.

to the TM that supervises one of these classes. The TM synchronizes T against other transactions in its class using a local mechanism similar to locking. To synchronize T against transactions in other classes, the TM uses the synchronization method(s) specified by the conflict graph analysis. These methods are called "protocols" and are described below.

3.3 Timestamp Based Protocols

To synchronize two transactions that conflict dangerously, one must be run first, and the other delayed until it can safely proceed. In locking systems, the execution order is determined by the order in which transactions request conflicting locks. In SDD-1, the order is determined by a total ordering of transactions induced by timestamps. Each transaction submitted to SDD-1 is assigned a globally unique timestamp by its TM. Timestamps are generated by concatenating a TM identifier to the right of the network clock time, so that timestamps from different TMs always differ in their low order bits. This means of generating unique timestamps was proposed in [THOMAS].

The timestamp of a transaction is attached to all Read and Write commands sent to DMs on its behalf. In addition, each Read command contains a list of classes that conflict dangerously with the transaction issuing the Read (this list was determined by the conflict graph analysis). When a DM receives a Read command, it defers the command until it has processed *all earlier* Write commands (i.e., those with smaller timestamps) and *no later* Write commands (i.e., those with larger ones) from the TMs for the specified classes. The DM can determine how long to wait because of a DM-TM communication discipline called *piping*.

Piping requires that each TM send its Write commands to DMs in timestamp order. In addition, the Reliable Network guarantees that messages are received in the order sent. Thus when a DM receives a Write from (say) TM_X timestamped (say) TS_X, the DM knows it has received *all* Write commands from TM_X with timestamps less than TS_X. So, to process a Read command with timestamp TS_R, the DM proceeds as follows:

> For each class specified in the Read command, the DM processes all Write commands from that class's TM up to (but not beyond) TS_R. If, however, the DM has already processed a Write command with timestamp beyond TS_R from one of these TMs, the Read is rejected.

To avoid excessive delays in waiting for Write commands, idle TMs periodically send null (empty) timestamped Write commands; also, an impatient DM can explicitly request a null Write from a TM that is slow in sending them.

The synchronization protocol we have just described roughly corresponds to locking, and is designed to avoid "race conditions" [BERNSTEIN et al. c]. However, there are several variations of this protocol, depending on the type of timestamp attached to Read commands and the interpretation of the timestamps by DMs. For example, read-only transactions can use a less expensive protocol in which the DM selects the timestamp, thereby avoiding the possibility of rejection and reducing delay. The variety of available synchronization protocols is an important feature of SDD-1's concurrency control.

The SDD-1 concurrency control mechanism is described in greater detail in [BERNSTEIN et al. a,b]; its correctness is formally proved in [BERNSTEIN and SHIPMAN].

When all Read commands have been processed, consistent private copies of the read-set have been set aside at all necessary DMs. At this point, the Read phase is complete.

4. Distributed Query Processing

Having obtained a consistent copy of a transaction's read-set, the next step is to compile the transaction into a parallel program and execute it. The key part of the compilation is Access Planning, an optimization procedure that minimizes the object program's inter-site communication needs while maximizing its parallelism. Access Planning is discussed in Section 4.1, and execution of compiled transactions is explained in 4.2.

4.1 Access Planning

Perhaps the simplest way to execute a distributed transaction T is to move all of T's read-set to a single DM, and then execute T at that DM. (See Figure 4.1.) This approach works but suffers two drawbacks: (1) T's read-set might be very large, and moving it between sites could be exorbitantly expensive; and (2) little use is made of parallel processing. Access Planning overcomes these drawbacks.

Simple Execution Strategy Figure 4.1

Given the database of Figures 2.2 and 2.3

```
Let transaction T5 be
   Range of C is CUSTOMER;
   Replace C (ChkBal=ChkBal+LoanBal) where LoanBal<0;
```

(The effect of T5 is to credit loan overpayments to customers' checking accounts.)

Simple strategy
  Move every fragment that could potentially contribute to T5's result to a designated site. Process T5 locally at that site.

The Access Planner produces object programs with two phases, called <u>reduction</u> and <u>final processing</u>. The reduction phase eliminates from T's read-set as much data as is economically feasible without changing T's answer. Then, during final processing, the reduced read-set is moved to a designated "final" DM where T is executed. This structure mirrors the simple approach described above, but lowers communication cost and increases parallelism via reduction.

Reduction employs the familiar <u>restriction</u> and <u>projection</u> operators, plus an operator called <u>semi-join</u>, defined as follows: let R(A,B) and S(C,D) be relations; the semi-join of R by S on a qualification q (e.g. R.B = S.C) equals the join of R and S on q, projected back onto the attributes of R. (See Figure 4.2.) If R and S are stored

Semi-Join Examples Figure 4.2

Given:

CUST(Name,	ChkBal,	LoanBal)
Jeff.	$300	$30000
Adams	$100	$20000
Polk	$250	$20000
Tyler	$100	$15000
Buchanan	$700	$40000
Johnson	$200	$20000

AUTO_PAY(Name,	Amount)
Jeff.	$300
Adams	$200
Polk	$200
Tyler	$150
Buchanan	$400
Johnson	$200

Example (i)

The semi-join of
CUST by AUTO_PAY on CUST.ChkBal=AUTO_PAY.Amount
equals the join of
CUST and AUTO_PAY on CUST.ChkBal=AUTO_PAY.Amount

(Name,	ChkBal,	LoanBal,	Name,	Amount)
Jeff.	$300	$30000	Jeff.	$300
Johnson	$200	$20000	Adams	$200
Johnson	$200	$20000	Polk	$200
Johnson	$200	$20000	Johnson	$200

projected onto

(Name,	ChkBal,	LoanBal)
Jeff.	$300	$30000
Johnson	$200	$20000.

Example (ii)

The semi-join of
CUST by AUTO_PAY on CUST.ChkBal<AUTO_PAY.Amount
and CUST.Name=AUTO_PAY.Name
equals:

(Name,	ChkBal,	LoanBal)
Adams	$100	$20000
Tyler	$100	$15000

(These are customers whose balances are insufficient for their automatic loan payments.)

at different DMs, this semi-join is computed by projecting S onto the attributes of q (i.e. S.C), and moving the result to R's DM.

We define the *cost* of an operator to be the amount of data (e.g., number of bytes) that must be sent between sites to execute it, while its *benefit* is the amount by which it reduces the size of its operand.* Restrictions and projections can be computed with no inter-site data transfer**, and always produce monotonically smaller relations. So, under our cost definition, such operations are always cost beneficial. Semi-joins, on the other hand, require inter-site data movement whenever their operands are stored at different sites. Hence the cost effectiveness of a semi-join depends on the database state. The problem of Access Planning is to construct a program of cost beneficial semi-joins, given a transaction and a database state.

* This definition is appropriate because communications is the bottleneck in a distributed DBMS.

** We ignore the cost of sending the restriction or projection command from TM to DM. In practice this cost is negligible.

The procedure we employ uses a hill-climbing discipline, starting from an initial feasible program and iteratively improving it. The initial program is essentially the simple approach described at the beginning of this section. The Access Planner improves this program by first adding all restrictions and projections permitted by T, and then iteratively incorporating cost-beneficial semi-joins. This process terminates when no further cost-beneficial semi-joins can be found. A final stage reorders the semi-joins to take maximal advantage of their reductive power, since the order in which semi-joins are selected may be suboptimal for execution.

4.2 Distributed Execution

The programs produced by the Access Planner are non-looping parallel programs and can be represented as data flow graphs [KARP and MILLER]. To execute the program, the TM issues commands to the DMs involved in each operation as soon as all predecessors of the operation are ready to produce output.

The effect of execution is to create at the final DM a temporary file to be written into the database (if T is an update) or displayed to the user (if T is a retrieval). At this point, the Execute phase has completed.

5. Reliable Writing

To complete transaction processing, the temporary file at the final DM must be installed in the permanent database and/or displayed to the user. Since the database is distributed, the temporary file is first split into a set of temporary files $F_1,\ldots,F_n$ that list the updates to be performed at each of $DM_1,\ldots,DM_n$ respectively; if any results must be displayed to the user, let us treat the user as one of the DMs.

Each of these temporary files is transmitted to the appropriate DM as a Write command. The problem is to ensure that failures cannot cause some DMs to install updates while causing others not to. We must protect against two types of failures: failure of a receiving DM, and failure of the sender; the former are handled by reliable delivery and the latter by transaction control, described in Sections 5.1 and 5.2 respectively. In addition, it is necessary to ensure that updates from different transactions are installed in the same "effective" order at all DMs. This problem is addressed in 5.3.

Ideally one would like 100% protection against failures, but this goal is theoretically unattainable [GRAY]. Instead our goal is to attain acceptably high levels of protection, and, moreover, to make the level of protection a database design parameter.

5.1 Guaranteed Delivery

Techniques for reliable message delivery are well-known in the communication field as long as both sites are up. For example, in ARPANET errors due to duplicate messages, missing messages, and damaged messages are detected and corrected by the network software. SDD-1's guaranteed delivery mechanism addresses the additional threat that the sender and receiver are not up simultaneously to exchange messages.

To solve this problem, the RelNet employs a mechanism called spoolers. A spooler is a process with access to secondary storage that serves as a first-in-first-out message queue for a failed site. Any message sent to a failed DM is delivered to its spooler instead. Each spooler manages its secondary storage using conventional DBMS reliability techniques [VERHOFSTAD] to guarantee the integrity of these messages. In addition, protection against spooler site failures is attained by employing multiple spoolers; as long as one spooler is up and running correctly, messages can be reliably stored.

Using reliable delivery, WRITE messages can be sent to failed DMs. When a failed DM recovers, it can receive its (spooled) WRITE messages to bring its database up to date.

5.2 Transaction Control

Transaction control addresses failures of the final DM that occur during the Write phase. Suppose the final DM fails after sending files $F_1,...,F_{k-1}$, but before sending $F_k,...,F_n$. At this point, the database is inconsistent, because $DM_1,...,DM_{k-1}$ reflect the effects of the transaction while $DM_k,...,DM_n$ do not. Transaction control ensures that inconsistencies of this type are rectified in a timely fashion.

The basic technique employed is a variant of "two-phase commit" [GRAY]. During phase 1, the final DM transmits $F_1,...,F_n$ but the receiving DMs do <u>not</u> install them yet. During phase 2, the final DM sends <u>Commit messages</u> to $DM_1,...,DM_n$, whereupon each DM_i does the installation. If some DM, DM_k say, has received F_k, but not a Commit, and the final DM has failed, DM_k consults the other DMs. If any have received a Commit, DM_k does the installation; if none have received Commits, none do the installation, thereby aborting the transaction.

This technique offers complete protection against failures of the final DM, but is susceptible to multi-site failures. Enhancements that offer arbitrarily high protection against multiple failures are described in [HAMMER and SHIPMAN].

5.3 The Write Rule

If transactions T_1 and T_2 complete execution at approximately the same time and have intersecting write-sets, a mechanism is needed to ensure that their updates are installed "in the same order" at all DMs. One way to do this is to attach the transaction's timestamp to each Write command, and require that DMs process Write commands in timestamp order. This technique introduces unnecessary delays however. It is possible to do better by timestamping the database as well as Write commands.

Every physical data item in the database is timestamped with the time of the most recent transaction that updated it. In addition, each Write command carries the timestamp of the transaction that generated it. When an update is committed at a DM, the following Write rule is applied: for each data item, X, in the Write command, the value of X is modified at the DM if and only if X's stored timestamp is less than the timestamp of the Write command. Thus "recent" updates (ones with big timestamps) are never overwritten by "older" ones. The net effect is the same as processing Write commands in timestamp order. This technique was originally suggested by [THOMAS].

A principal objection to this technique is the apparent high cost of storing timestamps for every data item in the database. However this cost is reduced to acceptable levels by caching the timestamps (see [BERNSTEIN et al. b]).

When updates are installed at all DMs the Write phase is completed. At this point the transaction has been fully processed.

6. Directory Management

SDD-1 maintains directories containing relation and fragment definitions, fragment locations, and usage statistics. Since TMs use directories for every transaction, efficient and flexible directory management is important. The main issues in directory management are whether or not to store directories redundantly, and whether directory updates should be centralized or decentralized. We have made these issues a matter of database design by treating directories as ordinary user data. This approach allows directories to be fragmented, distributed with arbitrary redundancy, and updated from arbitrary TMs.

But there are some problems. First, performance could be degraded by requiring that every directory access incur general transaction overhead, and by requiring that every access to remotely stored directories incur communication delays. We avoid these performance problems by caching recently referenced directory fragments at each TM, discarding them if rendered obsolete by directory updates. Since directories are relatively static, this solution is appropriate.

A second problem is that we now need a directory that tells where each directory fragment is stored. This directory is called the *directory locator*, and a copy of it is stored at *every* DM. This solution is appropriate because directory locators are relatively small, and quite static.

7. History

SDD-1 is the first general-purpose distributed DBMS ever developed. Its design was initiated in 1976 and completed in 1978. The first version of the system which included distributed query processing was released in mid-1978 and a complete prototype system including concurrency control and reliable writing will be released in autumn 1979. SDD-1 is implemented for DEC-10 and DEC-20 computers running the TENEX and TOPS-20 operating systems; its communication medium is the ARPA network. SDD-1 is built on top of existing software to the extent possible; most notably it employs an existing DBMS, called Datacomputer [MARILL and STERN], to handle all database management issues. The current system is configured with four sites, although the software can support any reasonable number.

The complete SDD-1 software consists of 25,000 lines of BCPL code. The compiled DM has 47K 36-bit words of code; the compiled TM has 120K 36-bit words of code, of which 45K words are "borrowed" from Datacomputer's object code. The design and implementation represents about 10 people years of effort.

8. Conclusion

SDD-1 is a general-purpose distributed DBMS, integrating database management, distributed processing, and reliable communication technologies into a cohesive system. This integration offers substantial benefits by combining the advantages of distributed processing with the advantages of centralized database management. At the same time it introduces new technical problems, of which the most critical are concurrency control, query processing, and reliable writing. This paper has outlined the SDD-1 solutions to each of these problems. The existence of SDD-1 as a system demonstrates that these problems can be solved in an integrated software system, and that distributed database management is indeed a feasible technology. For in-depth presentations of our techniques we refer the reader to [BERNSTEIN et al. a,b], [BERNSTEIN and SHIPMAN], [GOODMAN et al.] and [HAMMER and SHIPMAN].

Acknowledgements

The authors thank Carolyn Carleton, Emy Dickey and Nancy Wolfe for their editorial assistance in preparing this and the companion SDD-1 papers.

CONCURRENCY CONTROL IN SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES

PART I: DESCRIPTION

Philip A. Bernstein
David W. Shipman
James B. Rothnie

January 1, 1979

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Abstract

This paper presents the concurrency control strategy of SDD-1. SDD-1, a System for Distributed Databases, is a prototype distributed database system being developed by CCA. In SDD-1, portions of data distributed throughout a network may be replicated at multiple sites. The SDD-1 concurrency control guarantees database consistency in the face of such distribution and replication.

Table of Contents

1. Introduction

SDD-1 is a prototype distributed database system being designed and implemented at Computer Corporation of America. The system is designed to support databases that can be physically distributed with arbitrary redundancy over a network of hundreds of sites, while keeping data distribution and data redundancy invisible to the user. A principal problem of implementing systems of this type is maintaining the consistency of the database while concurrent user transactions attempt to update it. The concurrency control mechanism that SDD-1 uses to overcome this problem is the subject of this paper.

2. Literature Review

The concurrency control problem in database systems has been a major research focus for some time. In centralized database management systems (abbr. DBMSs), the conventional method to control concurrent update activity is two-phase locking [Eswaran et al.]. Two phase locking requires that every transaction:

1. locks the data it reads and writes before it actually accesses it, and
2. does not obtain any new locks after it has released a lock.

Once a data item is locked, no other transaction may lock that data item until the owner of that lock releases it. Research into locking-based concurrency controls has analyzed deadlock problems, logical locks described by predicates (instead of by data item names), granularity of locks, and efficient locking algorithms [Chamberlin et al.], [Eswaran et al.], [Gray et al.], [King and Collmeyer], [Reis and Stonebraker].

Locking methods have also been proposed for distributed DBMSs. One technique, called primary-site, uses a central lock controller to manage the locks [Alsberg and Day]. Alternatively, locks can be distributed with the data. Since data can be distributed redundantly, in principle all copies would have to be locked. To reduce locking overhead, one copy of each file (say) can be designated to be primary. Only the primary copy then needs to be locked, independent of which copies or how many copies are accessed [Stonebraker]. Variations of locking which set "imaginary locks" [Thomas a,b] or which use version numbers [Stearns et al.], [Reed], [Rosenkrantz et al.] have also been proposed. (See also [Bernstein and Shipman b] for a proof that these methods are essentially locking approaches.)

These distributed locking approaches are quite similar to centralized concurrency controls, with the usual termination problems of indefinite postponement and/or deadlock. These mechanisms do differ, however, from centralized schemes in one respect -- the possibility of asynchronous failures of sites and communication links while an update is in the midst of being processed. Many of the proposed distributed concurrency controls have concentrated on this problem of failure (e.g., [Alsberg and Day] [Menasce et al.] [Stonebraker] [Thomas]).

The concurrency control mechanism of SDD-1 differs from all of the above mechanisms in at least one way. In SDD-1, information about how transactions can conflict is preanalyzed before the transactions are submitted, so that not all transactions need synchronization. This preanalysis technique is the heart of the SDD-1 concurrency control and is the main topic of this paper. Also, the run-time synchronization mechanisms of SDD-1, which differ considerably from locking, are discussed. An early restricted version of the SDD-1 concurrency control is discussed in [Bernstein et al.].

This paper is organized in fifteen sections. We begin, in Section 3, with a review of those aspects of SDD-1 architecture that impact concurrency control. Section 4 defines correctness for a concurrency control mechanism. Then, in Sections 5 and 6, we discuss two important techniques on which the SDD-1 concurrency control is based: timestamps and transaction classes. Sections 7-10 develop the preanalysis technique. An overview of the mathematics used in preanalysis has been isolated in Section 9 and can be skipped without loss of continuity. Sections 11 and 13 describe implementation aspects of the mechanism, and Section 12 describes a special protocol for transactions that would otherwise induce tremendous synchronization overhead. In Section 14, we discuss the reliability aspects of the implementation. We conclude in Section 15 with a summary of the advantages of our method and a comparison with other proposed distributed concurrency controls.

3. Review of SDD-1 Architecture

The architecture of SDD-1 is described in [ROTHNIE and GOODMAN] and [ROTHNIE et al.]. We review here those aspects of the architecture that are needed for understanding the concurrency control mechanism.

A user of SDD-1 sees a conventional DBMS. The logical database is expressed in a relational data model which, from the viewpoint of the user's transaction, is nonredundant and nondistributed. Issues that are consequences of physical data distribution and redundancy are entirely handled by the system and are visible to the user transaction only insofar as they affect performance. Transactions are expressed as a program written in a semi-procedural data manipulation language called Datalanguage [CCA].

Internally, SDD-1 consists of two types of modules, called transaction modules (abbr. TMs) and data modules (abbr. DMs). Each site can contain either one or both types of modules. DMs store physical data and behave much like conventional (i.e., nondistributed) DBMSs. TMs are responsible for supervising the execution of user transactions, translating from the user's nondistributed view of the data to the realities of its distribution and redundancy.

The basic unit of user computation in SDD-1 is the transaction. The execution of each transaction is supervised by a TM and consists of three sequential steps:

1. The transaction reads a subset of the database, called its read-set, into a distributed private workspace.
2. It does some computation on the workspace.
3. The transaction writes some of the values in its workspace into a subset of the database, called its write-set. The write-set need not be included in the read-set.

Since the transaction is coded in terms of the logical database, and since the physical database in general has redundant copies of many logical data items, the TM must choose which physical copies of the logical data items referenced by the transaction should be read or written. To keep the physical database internally consistent, the TM must apply each write operation on a data item to all physical copies of that data item. However, only one of the physical copies of each logical data item needs to be used for reading.

To obtain the read-set data for a transaction's input and later to write its output into copies of its write-set, a TM sends READ and WRITE messages to DMs. A READ message is a request by a TM to read some of the data items stored at a DM and to store them in a local workspace at that DM on behalf of some transaction. A WRITE message is sent by a TM to a DM to report updates produced by a transaction which the TM supervised.

To process a transaction, a TM must send READ messages to obtain the transaction's read-set. Logical data items are obtained from physical copies selected by the TM. The TM sends a READ message to those DMs that store the selected copies to be read by the transaction.

After all READ messages have been processed (i.e., after they have been positively acknowledged), the TM supervises the execution of the transaction. This function of the TM is performed by the access planner and is described in [WONG et al]. It is the job of the concurrency control mechanism to guarantee that the physical read-set obtained by READ messages is internally consistent, so that the transaction will produce correct output.

Write operations performed by the transaction are put into a temporary file and are deferred until the transaction completes execution. After the transaction completes execution, the TM broadcasts these updates to DMs as WRITE messages. Each update to a logical data item, say x, is sent to all DMs that have a stored copy of x.

A TM sends at most one READ message and at most one WRITE message to each DM on behalf of a single transaction. If, for example, a transaction reads data from two data items that reside at the same DM, then only one READ message is issued to read both data items. This is an important point, as each DM performs READs and WRITEs as atomic operations; for example, none of the data read by a READ message can be updated by some WRITE while the READ is being processed.

4. Concurrent Correctness

The system usually has many transactions in progress at any one time, both because there are multiple TMs operating concurrently within the system and because individual TMs are processing transactions concurrently. If the READs and WRITEs that implement these transactions were arbitrarily interleaved, then serious problems of database consistency would result. The usual method of avoiding these consistency problems is by guaranteeing that the execution of transactions is serializable [ESWARAN et al] [PAPADIMITRIOU et al] [ROSENKRANTZ et al].

We say that an interleaved execution of a set of transactions is serializable if it is "equivalent" to a history of operation in which each of the transactions runs alone to completion before the next one begins. Two executions are equivalent if in both executions each transaction produces the same output, thereby leading to the same final state of the database. That is, an interleaved execution is serializable if it could be reproduced by a non-interleaved (i.e., serial) execution of the same set of transactions. Note that serializability requires only that there exists some serial order equivalent to the actual interleaved execution. There may in fact be several such equivalent serial orderings.

The adoption of serializability as the criterion for concurrent correctness is based on the assumption that each user transaction will preserve database consistency if it runs atomically. That is, if only one transaction is allowed to execute at a time, and if the database state is initially consistent, then after executing a transaction the database state must still be consistent. So, a serial ordering of transaction executions will, by induction, result in a consistent database state. Since a serializable execution is equivalent to some serial one, a serializable execution results in a consistent database state as well.

The issue of serializability arises because a system's atomic actions are at a finer granularity than its users' atomic actions. In SDD-1, the users' atomic actions are transactions, while the system's atomic actions are the execution of READ and WRITE messages at the DMs. Each DM behaves as if READs and WRITEs are processed atomically, so it is impossible for a READ operation to observe the effects of only a part of a WRITE operation at a DM.

When a system allows the execution of several transactions at the same time, then the system's operations corresponding to different transactions are interleaved. If the interleaving is not controlled, there is no guarantee that the behavior of such a system conforms to the user's expectation that each transaction is processed as an indivisible computation.

For example, assume there is a single copy of data item x, which initially has the value x=0. There are two transactions in the system; transaction i sets x:=x+1, and transaction j sets x:=x+2. The following sequence of events occurs:

Transaction i reads x=0

Transaction j reads x=0

Transaction j sets x:=2

Transaction i sets x:=1

Any serial execution of the two transactions, one after the other, would have resulted in setting x to 3. However, the result of this interleaved execution is to set x to 1, contrary to the user's intention. This execution history is not serializable, since no serial processing of these transactions will produce the observed effects.

To guarantee serializability in SDD-1, we apparently need to avoid undesirable interleavings of READ and WRITE messages -- those that lead to nonserializable executions. We accomplish this goal using two mechanisms. First, we examine each transaction to determine if it is conceivable that it could participate in a nonserializable execution. As we will see, many transactions will never produce READs and WRITEs that interleave badly with other transactions, and hence can be run unsynchronized. Second, for those transactions that are determined to be dangerous because they can participate in nonserializable executions, we synchronize their READ and WRITE messages using protocols that avoid undesirable interleavings. These protocols are based on a timestamping mechanism and are quite different from the locking protocols used in conventional centralized DBMSs.

As we will see, most of the effort in distinguishing transactions that require no synchronization from the dangerous ones is done statically when the database is designed. When a transaction is actually submitted, a simple local table look-up is sufficient to determine how much, if any, synchronization is required. The run-time mechanism is the collection of protocols that must be invoked for those transactions that do require synchronization.

Note that these two components of the concurrency control mechanism are independent. Our technique for analyzing transactions to determine sources of nonserializability could be used in conjunction with conventional locking protocols. Or, we could run <u>all</u> transactions using our timestamp-based protocols and ignore the preanalysis step entirely, as in present systems that use locking without preanalysis. Together the two mechanisms provide a powerful technique for synchronizing concurrent transactions at low cost.

Before describing the heart of the system -- the method for determining the amount of synchronization required by each transaction and the protocols that effect that synchronization -- we must first describe two basic concepts that underlie much of the concurrency control mechanism. These concepts, timestamps and transaction classes, are described in the next two sections.

5. Timestamps

Each transaction executed by SDD-1 is assigned a globally unique timestamp. Transaction timestamps serve a number of purposes for synchronizing READs and WRITEs. To generate globally unique timestamps, a TM reads its local clock and appends its unique TM number as the low order bits of the timestamp. By requiring that once a clock is read it cannot be read again until it has been incremented, we ensure that every timestamp is globally unique within the system [THOMAS a].

The clocks are actually maintained as part of the Reliable Network, the reliable communications facility of SDD-1. By using the clock synchronization method described in [LAMPORT], the system behaves as if there were a single virtual clock available to all sites.

One use of timestamps is in processing WRITE messages that arrive at a DM out of order. The problem is that the WRITE messages sent by two transactions that update the same logical data item may be processed in different orders at different DMs, thereby producing mutually inconsistent copies of the data item. One way to solve this problem is to attach the transaction's timestamp to all of its WRITE messages, and then require that WRITE messages be processed in timestamp order at all DMs. A better method that gives more flexibility to DMs in the processing of WRITE messages uses timestamped data items and is adopted in SDD-1 (this method was originally suggested in [THOMAS a]).

A transaction's timestamp is carried on all of its WRITE messages. In addition, every physical data item at every DM has an associated timestamp. Note that timestamps are attached to physical data items; there may be many physical copies of a logical data item and each one has its own attached timestamp. The timestamp of a data item is the timestamp of the last WRITE message that updated it. Each DM processes WRITE messages according to the following WRITE message rule: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the WRITE message's timestamp. (Recall that a WRITE message contains the final values of data items, not computations to be performed on them.) So, to process a data item in a WRITE message, the DM compares the timestamp of the WRITE message with the timestamp of its stored copy of the data item. If the timestamp of the WRITE message exceeds the timestamp of the stored data item, then the new value of the data item in the WRITE message is written into the stored data item along with the new timestamp. Otherwise, the update is not performed on that stored data item. This is a data item by data item check; some data items in the WRITE message may result in update operations while others may not.

Whenever a WRITE message for a recent transaction that updates some data item is processed at a DM before a WRITE message for an earlier (i.e., older) transaction that updates the same data item, the latter WRITE message will contain a data item update that is not performed. Such a situation is not an error. It is simply the way that the system reorders updates to occur in the same order that their generating transactions executed. That is, the net effect of a set of WRITE messages processed at a DM in arbitrary order is the same as the effect of processing them in timestamp order without the WRITE message rule. The principal advantage of using the WRITE message rule is that WRITE messages can be processed as soon as they are received, thereby avoiding artificial queuing delays at the DMs.

Note that the correctness of the WRITE message rule in reordering updates does not require that clocks in different TMs be at all synchronized. This is true of other timestamp related mechanisms in SDD-1 as well. For reasons of efficiency, however, it is necessary to assume that clock values in different TMs are reasonably close to each other.

A principal objection to timestamped data items is its cost. However, not all timestamps actually need to be stored. If the timestamp of a data item is earlier than the timestamp of any transaction whose WRITE messages have not yet been processed, then the data item's timestamp is effectively zero. Any WRITE message that tries to update that data item will succeed, because the WRITE message will have a later timestamp than the data item. So, we need only maintain the timestamps of recently updated data items. If a data item is not updated for a while (say a few minutes), then its timestamp can be assumed to be zero and therefore dropped. A caching mechanism for timestamps using differential files is used in SDD-1 for this purpose. Using this mechanism, we judge that the overhead in maintaining timestamps will be small, since only a small portion of the data items will require their timestamps to be stored in the cache at one time.

6. Transaction Classes

A crucial aspect of the SDD-1 concurrency control mechanism is its ability to distinguish between transactions that require synchronization and those that do not. By examining the read-set and write-set of transactions, the system can determine which transactions conflict with each other. Intuitively, two transactions _conflict_ if the read-set or write-set of one intersects the write-set of the other. Such conflicts are the main cause of nonserializability. They are avoided in conventional DBMSs by locking data items so that two conflicting transactions never run concurrently. However, preventing _all_ conflicts is more than what is required to guarantee serializability. By analyzing a graph theoretic representation of the transactions, called a _conflict graph_, the system can isolate the dangerous conflicts that can potentially lead to nonserializability. This analysis technique will be described in detail later in the paper.

Unfortunately, analyzing the conflict graph at run-time for all executing transactions is too time consuming. Also, since the transactions are distributed at run-time, assembling a conflift graph would require too much communication. So, we transform this run-time analysis into a static analysis done only once at database design time by capitalizing on the predictability of transaction types in the following way.

When designing the database, the database administrator establishes a static set of transaction classes. Formally, each transaction class is defined by a logical read-set and write-set and is assigned to run at a particular TM. A transaction fits in a class if the read-set and write-set of the transaction is contained (respectively) in the read-set and write-set of the class. Read-set and write-set definitions are expressed using simple predicates, so that class membership can be checked quickly (see Figure 6.1).

The conflict graph analysis is now done on the statically defined transaction classes instead of on the transactions themselves. This analysis yields the type of synchronization, if any, required for each class. At run-time, when a transaction is submitted to a TM, the TM selects a class in which the transaction fits and applies the type of synchronization specified by the analysis for that class.

Class Definitions Using Simple Predicates Figure 6.1

Relation Schema: INVENTORY (ITEM#,DESCRIPTION,PRICE, QUANTITY)

Class 1
read-set: INVENTORY [ITEM#,PRICE]
write-set: INVENTORY [PRICE]
comments: transactions that update prices

Class 2
read-set: INVENTORY [ITEM#,QUANTITY] WHERE (PRICE > $100)
write-set: INVENTORY [QUANTITY]
comments: transactions that update quantities of high-priced items

Class 3
read-set: INVENTORY [ITEM#,DESCRIPTION,PRICE] WHERE (QUANTITY > 0)
write-set: user's terminal
comments: transactions that display item information about items currently in stock.

The utility of classes lies in the property that two transactions that run in different classes conflict only if their classes conflict. Hence, conflicts between transactions can be determined by conflicts between classes. So, an analysis of the classes at database design time is sufficient to determine potentially dangerous conflicts between transactions at run time. We believe that, for many kinds of applications, the most frequent determination will be that the class participates in no dangerous conflicts and can therefore run with only local synchronization.

How a TM Processes a transaction   Figure 6.2

```
Do Forever;
   Wait for a transaction, T, to arrive;
   Find a class, C, in which T fits;
   If C cannot be processed locally
    then forward T to a site that can process C
    else begin
         look up the synchronization rules for class C,
         send out appropriate READ messages on
             behalf of T, synchronizating where necessary;
         supervise the distributed execution of T;
         send out WRITE messages on behalf of T
         end
   end
```

For a set of class definitions to be feasible, it must cover *all* transactions that might ever be submitted. It is *not* necessary that *every* TM have enough classes to accept all possible transactions, since a TM can forward a transaction to some other TM for execution. However, it *is* necessary that every possible transaction fit in a class supported by *some* TM. A sketch of how a transaction is routed and executed by TMs appears in figure 6.2.

7. Synchronizing Transactions Within a Class

To ensure the serializability of transactions which execute in the same class, we require that within a class all of the transactions are actually executed serially, one after another. To formalize this requirement, some notation is helpful. Let the processing of a READ message on behalf of transaction i at DM_{alpha} * be denoted R^{i}_{alpha}. Similarly, let the processing of a WRITE message on behalf of transaction i at DM_{alpha} be denoted W^{i}_{alpha}. Then we can express the requirement that transactions within a class run serially as follows:

Class Pipelining Rules: For each DM_{alpha}, for each class $\bar{I}$, and for each pair of transactions i_1 and i_2 in $\bar{I}$,

C1. If i_1 and i_2 both read from DM_{alpha}, then $R^{i_1}_{alpha}$ is processed before $R^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

* We use lower case Greek letters to denoted DMs. We use lower case Roman letters i,j,k,... to denote transactions. We denote the class in which transaction i executes by $\bar{I}$.

C2. If i_1 and i_2 both write into DM_{alpha}, then $W^{i_1}_{alpha}$ is processed before $W^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

C3. If i_1 reads some data item at DM_{alpha} and i_2 writes some data at DM_{alpha}, then $R^{i_1}_{alpha}$ is processed before $W^{i_2}_{alpha}$ only if i_1 has an earlier timestamp than i_2.

The class pipelining rules force transactions that run in a single class to be processed serially at all DMs in the same order. Rules C1 and C2 guarantee that READ and WRITE messages (respectively) from each class are processed in timestamp order at all DMs. Rule C3 guarantees that READ messages from each class only see updates from earlier WRITE messages in that class.* These rules are sufficient to guarantee the noninterference of any two transactions that run in a single class.

The class pipelining rule, although stated in terms of DMs, is actually enforced by mechanisms at both TMs and DMs. For each class that a TM processes, the messages from that class are sent to each DM in an order that is

* Actually, a weaker condition than C3 is possible. C3 must only be applied when the read-set of i_1 at DM_{alpha} intersects the write-set of i_2 at DM_{alpha}, since this is the only case when i_1 can actually see the update produced by i_2. However, to eliminate several special analyses that would be required, we assume C3 is always applied.

consistent with C1-C3. The communications network (ARPANET, in our case) guarantees that messages are received in the order they were sent, for any point-to-point communications channel. The DMs process messages within a class in the order in which they are received, thereby enforcing C1-C3.

8. Interclass Interference

8.1 An Example of Safe Interference

We say that a set of transactions *interfere* if the system allows them to be interleaved in a nonserial manner. Given the class pipelining rule, we need not be concerned with interference among transactions in the same class, since they are run serially. The problem now is to avoid interference among transactions in different classes. A critical aspect of our solution to this problem is isolating those cases where transactions in different classes never interfere with each other. This requires some subtlety, for even when transactions read and write the same data items, they may not interfere, as illustrated by the following simple example.

Suppose we run two transactions, say i and j, in two different classes, $\bar{i}$ and $\bar{j}$, each of which first finds the EMPLOYEE record whose NAME domain has the value 'JON DOE'; then each writes a distinct new value into the PHONE#

domain of that record (the phone numbers written by the two transactions are different). Naturally, the final value of JON DOE's PHONE#, after both transactions execute, is dependent on the order in which their write operations were processed. However, no matter how their read and write operations are interleaved, the execution will be serializable. The transactions will always appear to have executed serially with the order of their writes determining the order of the transactions in the serialization; the transaction that writes JON DOE's PHONE# first appears first in the serialization. Therefore, even though the transactions have overlapping write-sets -- a situation that conventionally requires locking -- no synchronization is necessary.

To exploit situations of this type, we must determine safe patterns of interleaved reads and writes that require no synchronization. This determination is accomplished by analyzing conflicts between transaction classes. For example, an analysis of classes $\bar{I}$ and $\bar{J}$ above would show that all patterns of interleaved reads and writes are serializable. This analysis is performed on a graph theoretic representation of transaction conflicts, and is the subject of the next section.

8.2 Conflict Graphs

As we observed in Section 6, two transactions from different classes conflict only if their classes conflict.

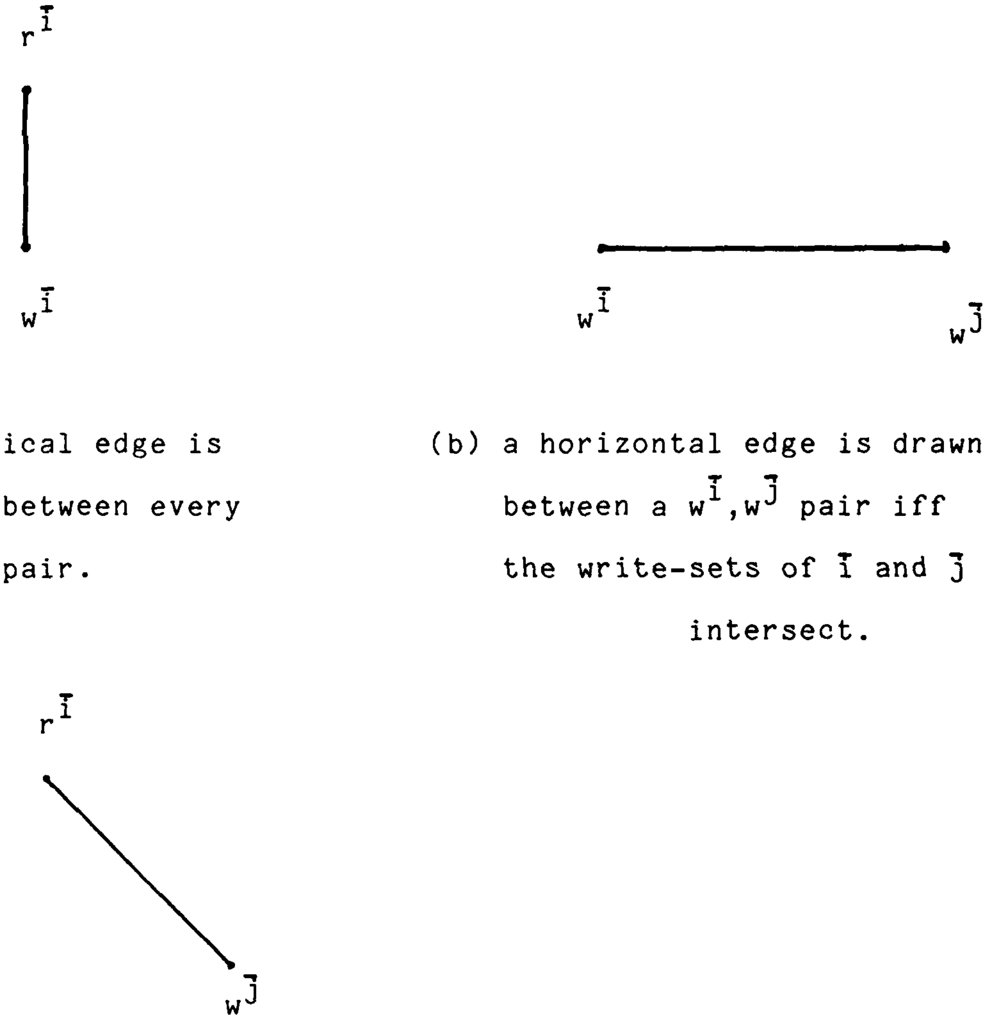
To formalize this, we say that WRITE message W^{i}_{alpha} conflicts with a READ message R^{j}_{alpha} iff transaction i's write-set intersects transaction j's read-set. A WRITE message W^{i}_{alpha} conflicts with another WRITE message W^{j}_{alpha} iff transaction i's write-set intersects transaction j's write-set. It follows that if R^{i}_{alpha} conflicts with W^{j}_{alpha}, then the read-set of class $\bar{I}$ intersects the write-set of class $\bar{J}$. By examining class conflicts, we can predict potential transaction conflicts, which are a primary component of the serializability problem. It will turn out that this examination of class conflict will lead us to our goal -- a method for determining the amount of synchronization required by each transaction.

The method begins with the construction of a conflict graph (see Figure 8.1). In the graph, each class, say $\bar{I}$, is modeled by two nodes labelled $r^{\bar{I}}$ and $w^{\bar{I}}$. For each class, $\bar{I}$, an edge $<r^{\bar{I}},w^{\bar{I}}>$ connecting them is drawn (Figure 8.1a). When the write-sets of two classes, say $\bar{I}$ and $\bar{J}$, intersect, then the edge $\langle w^{\bar{i}}, w^{\bar{j}} \rangle$, called a horizontal edge, is drawn (Figure 8.1b). Similarly, if the read-set of one class (say $\bar{i}$), intersects the write-set of another class (say $\bar{j}$), then an edge $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ called a diagonal edge is drawn (Figure 8.1c).

For a given set of classes, $\underline{C}$, we denote the conflict graph for $\underline{C}$ by $CG_{\underline{C}}$. A sample conflict graph appears in Figure 8.2.

We will use the conflict graph to help us predict the amount of synchronization required by each transaction class. The connection between synchronization protocols and conflict graphs is developed in Section 9. Since this development is lengthy and may not be of interest to all readers, we summarize the principle results of Section 9 in Section 10. Hence Section 9 can be skipped, if desired, without loss of continuity.

Conflict Graph Edges Figure 8.1



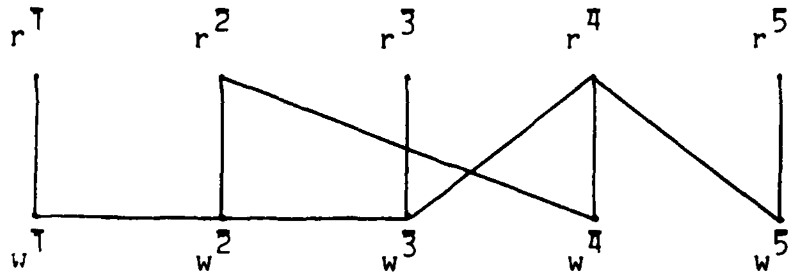
(a) a vertical edge is drawn between every r^I,w^I pair.

(b) a horizontal edge is drawn between a w^I,w^J pair iff the write-sets of I and J intersect.

(c) a diagonal edge is drawn between an r^I,w^J pair iff the read-set of I intersects the write-set of J.

A Sample Conflict Graph

Figure 8.2



9. Conflict Graph Analysis

9.1 Serializing Logs

Depending on the order in which READ and WRITE messages are processed by the system, an interleaved execution of transactions may or may not be serializable. To understand which message orderings are serializable, we need a notation that models these orderings. In our notation, we will represent the ordered processing of READ and WRITE messages at a DM by a log. A log is simply a string of R's and W's that have the same DM subscript. For example, R^1_{alpha} W^2_{alpha} W^1_{alpha} R^5_{alpha} W^5_{alpha} R^4_{alpha} is a log describing the order in which READ and WRITE messages were processed at DM_{alpha}. When we say, for example, that R^i_{alpha} precedes W^j_{alpha} (in DM_{alpha}'s log), we mean that R^i_{alpha} was processed before W^j_{alpha} at DM_{alpha}.

A log is a complete representation of the computations performed on the database at a DM. If we were given the list of data items read and written by each WRITE message as well as the timestamps of transactions (so that we could correctly apply the WRITE message rule), then we would be able to reproduce the computation that was actually performed at the DM. So, an "interleaved execution of transactions" in SDD-1 is modelled by a "collection of DM logs, one per DM". We will therefore use these two terms interchangeably.

Suppose we are given an interleaved execution of N transactions, represented by a set of DM logs. Which of the N! possible serializations of the transactions is an equivalent serialization of the given logs? A serialization is equivalent to the given logs if that serial execution of the transactions on a nondistributed, nonredundant database (represented by the serialization) produces the same computation as the interleaved execution on the distributed, redundant database (represented by the DM logs). It is a theorem that if each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were applied to it in the given interleaved execution, then each transaction will perform the same computation in the serialization as it did in the given interleaved execution [Papadimitriou et al]. We can guarantee this condition by requiring that the serialization satisfy the following three rules: For each i,j, and alpha

1. If W^{i}_{alpha} precedes and conflicts with R^{j}_{alpha}, then i must precede j in the serialization;
2. If R^{j}_{alpha} precedes and conflicts with W^{i}_{alpha}, then j must precede i in the serialization;
3. If W^{i}_{alpha} conflicts with W^{j}_{alpha}, then i and j must appear in the serialization in their timestamp order.

If the serialization obeys (1) and (2), then write operations in the serialization precede exactly the same read operations as they did in the given interleaved execution. However, this is not the same as saying that each transaction reads from a database that has had exactly the same write operations applied to it in the serialization as were _applied_ to it in the given execution. The reason is that due to the WRITE message rule, the order in which WRITE messages are _processed_ is not the same as the effective order in which the write operations are _applied_ to the database; indeed, some write operations are not applied at all. To understand this subtle distinction is to understand the need for rule (3).

In the logs, the WRITE message rule prevents certain write operations from being applied; this occurs when a WRITE message with an early timestamp arrives after a WRITE message with a later timestamp and both WRITE messages write into a common data item. The WRITE message rule is an artifact of the distributed execution of SDD-1, and would not have been applied if the transaction were executed serially on a nondistributed, nonredundant database. In essence, this means that the serialization must produce the same computation without the WRITE message rule that the given logs produced with the WRITE message rule. Rules (1) and (2) alone are not strong enough to make this guarantee.

For example, suppose the log for DM_{alpha} contains W^{i}_{alpha} W^{j}_{alpha} R^{k}_{alpha} where j has an earlier timestamp than i and all three messages write into or read from data item x. The WRITE message rule prevents W^{j}_{alpha} from overwriting x, so R^{k}_{alpha} reads x from W^{i}_{alpha}. We want the same relative ordering of R^{k}_{alpha} and W^{i}_{alpha} to appear in the serialization. So, transaction j must precede transaction i in the serialization. However, the serialization [i,j,k] would be permitted by the rules (1) and (2) alone; this is incorrect because transaction k would read x from j (not i) in this serialization.

Rule (3) guarantees that write operations in the serialization are applied in the same relative order as they are applied in the given logs. It "factors out" the WRITE message rule from the serialization by requiring the write operations to appear in the order that they were effectively applied, rather than the order in which they were processed.

By developing rules (1) - (3), we have related the order of conflicting READ and WRITE messages in DM logs to the order of transactions in serializations. As we know, not all interleaved executions are serializable. So, as we would expect, there are DM logs that have no serialization that obeys rules (1)-(3). In principle, we could schedule READ and WRITE messages by continually checking rules (1)-(3) at run-time so that the order in which READ and WRITE messages are processed can always be serialized. However, this would be very costly in computation time and communication traffic. Instead, we use the conflict graph model of transaction conflicts to guide us in synchronizing READ and WRITE messages so that a serialization obeying rules (1)-(3) is always possible.

The conflict graph is used tc determine potentially nonserializable executions of conflicting transactions. The interpretation of diagonal and horizontal edges can be used to extend rules (1) - (3): For each i, j, and alpha

1'. If $\langle w^{\bar{i}}, r^{\bar{j}} \rangle$ is a diagonal edge of CG and W^{i}_{alpha} precedes R^{j}_{alpha} in DM_{alpha}'s log, then i must precede j in any serialization.

2'. If $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ is a diagonal edge of CG and R^{i}_{alpha} precedes W^{j}_{alpha} in DM_{alpha}'s log, then i must precede j in any serialization.

3'. If $\langle w^{\bar{i}}, w^{\bar{j}} \rangle$ is a horizontal edge of CG, then i and j must appear in the serialization in their timestamp order.

Since two transactions conflict only if their classes conflict, any serialization that satisfies (1')-(3') will satisfy (1)-(3) as well. The advantage to using (1') - (3') in place of (1) - (3) is that the former are stated entirely in terms of class conflicts, which are known in advance.

In SDD-1, there is always a serialization of the executed transactions that satisfies (1')-(3'). The mechanisms that are used to guarantee that such a serialization always exists are called protocols.

9.2 Protocol P1 and the Acyclicity Theorem

To understand why we need protocols, let us consider a system consisting of two classes, say $\bar{I}$ and $\bar{J}$, such that only one transaction is processed in each class, say transactions i and j. Under what conditions will these two transactions be serializable? If there are no horizontal or diagonal edges connecting $\bar{I}$ and $\bar{J}$ in the conflict graph, then (1')-(3') are trivially satisfied. In this case, i and j are serializable; in fact, either serialization will do. What if $\bar{I}$ and $\bar{J}$ are connected by some edge?

If $<w^{\bar{I}},w^{\bar{J}}>$ appears in CG, and if W^{i}_{alpha} and W^{j}_{alpha} are processed (for some DM_{alpha}), then according to rule (3') i and j must be serialized in timestamp order. If this is the only edge connecting $\bar{I}$ and $\bar{J}$, then the transactions are still surely serializable. For no matter how many DMs process WRITE messages from both transactions, each DM will apply the WRITE message rule, thereby making it look like i was processed before j. Therefore, applying rule (3') at all DMs will yield the same requirement that i and j be serialized in the same timestamp order. The only way we could get into trouble is if one DM believes i should precede j in the serialization while another believes j should precede i -- a clear impossibility using (3'). So, if $\langle w^{\bar{i}}, w^{\bar{j}} \rangle$ is the only edge connecting $\bar{i}$ and $\bar{j}$, we are safe.

If $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ appears in CG, then we have a potential problem. Suppose W^{j}_{alpha} precedes and conflicts with R^{i}_{alpha} and R^{i}_{beta} precedes and conflicts with W^{j}_{beta}. Rule (1') applied at DM_{alpha} says that j should precede i while rule (2') applied at beta says that i should precede j. Since both cannot be simultaneously satisfied, we have a nonserializable interleaving. Apparently, we must introduce some synchronization mechanism to avoid this problem produced by the diagonal edge.

Protocol P1 is the mechanism used to synchronize diagonal edge conflicts. We say that transaction i obeys protocol P1 with respect to transaction j if the relative ordering of READ messages from i and WRITE messages from j are the same at all DMs where both appear and conflict. Stated more formally, if R^{i}_{alpha} precedes (resp. follows) and conflicts with W^{j}_{alpha} at DM_{alpha}, then if R^{i}_{beta} and W^{j}_{beta} conflict and both are processed at DM_{beta}, R^{i}_{beta} must precede (resp. follow) W^{j}_{beta} at DM_{beta}.

We require that if $\langle r^{\bar{I}},w^{\bar{J}}\rangle$ is an edge in CG, then for each pair of transactions i in class $\bar{I}$ and j in class $\bar{J}$, i must obey protocol P1 with respect to j. If the protocol is obeyed, then the nonserializable situation due to the opposite serializations implied by rules (1') and (3') cannot occur. Since this is the only problem a single diagonal edge can cause, P1 is sufficient to synchronize diagonal edges.

The above observations regarding single edge conflicts between two classes generalize directly to paths of conflicts. Suppose there is a single edge conflict between $\bar{I}$ and $\bar{k}$, and another one between $\bar{k}$ and $\bar{J}$. Again, assume one transaction runs in each class, say i, j, and k. Rules (1')-(3') only restrict the order of serialization between pairs of conflicting transactions. They will either require that i and j have a defined relative ordering (i.e., either i precedes k and k precedes j or i follows k and k follows j) or that they have no special required order (i.e., either i precedes k and j precedes k or i follows k and j follows k). In either case, the three transactions are serializable.

The only way the transactions might not be serializable is if there were two different paths from $\bar{I}$ to $\bar{J}$. Then, one path could lead to i preceding j according to rules

(1')-(3'), while the other path could lead to i following j. If this occurred, then the execution would be nonserializable. But note that it can only occur if there are *two* distinct paths. Two distinct paths that link $\bar{I}$ to $\bar{J}$ constitute a cycle. So, as long as there are no cycles in the conflict graph and each class runs one transaction, P1 is sufficient to guarantee serializability.

The class pipelining rule requires that transactions within a single class essentially run serially. So, the above statement about acyclic conflict graphs generalizes to the case of multiple transactions per class. (A proof of this fact is nontrivial and appears in [BERNSTEIN and SHIPMAN a].)

Our observations in this section can now be stated more formally as follows:

Acyclicity Theorem For a given set of transaction classes, $\underline{C}$, if

1. $CG_{\underline{C}}$ has no cycles, and
2. all classes in $\underline{C}$ obey the class pipelining rule, and
3. for each diagonal edge $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ in $CG_{\underline{C}}$ and transactions i in $\bar{I}$ and j in $\bar{J}$, transaction i obeys P1 with respect to j,

then all possible interleavings of transactions in classes in $\underline{C}$ are serializable.

To make the acyclicity theorem effective, we need to demonstrate an implementation for P1. This we will do in Section 11. First, however, we will show how to synchronize nonserializable situations caused by cycles.

9.3 Cycles, P3, and the Serializability Theorem

We have shown that if no cycles exist in the conflict graph and if P1 is properly applied, then all possible interleaved executions of transactions will be serializable. We also observed that cycles in the conflict graph can cause a nonserializable execution. If two distinct paths exist between two classes, $\overline{I}$ and $\overline{J}$, then the paths may lead to opposite serializations of transactions i in $\overline{I}$ and j in $\overline{J}$ according to rules (1')-(3') -- a nonserializable situation. To eliminate this possibility, we introduce a protocol that forces any two paths between $\overline{I}$ and $\overline{J}$ to always lead to the same relative ordering of i and j in all serializations. To illustrate the problem and the protocol that solves it, let us consider another example.

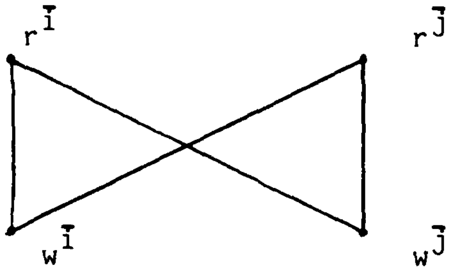
This time, suppose the database has one data item, x, stored at DM_{alpha}. Classes $\bar{i}$ and $\bar{j}$ both read from and write into x; for example, they both run transactions that increment x. The conflict graph for these classes contains two distinct edges, $<r^{\bar{i}},w^{\bar{j}}>$ and $<w^{\bar{i}},r^{\bar{j}}>$, connecting $\bar{i}$ and $\bar{j}$. These two edges together with $<r^{\bar{i}},w^{\bar{i}}>$ and $<r^{\bar{j}},w^{\bar{j}}>$ constitute a cycle (see Figure 9.2). The problem is that the diagonal edges may force opposite serializations of transactions in $\bar{i}$ and $\bar{j}$.

Consider, for instance, transactions i in $\bar{i}$ and j in $\bar{j}$ which execute their READ and WRITE messages in the following order: R^{i}_{alpha} R^{j}_{alpha} W^{i}_{alpha} W^{j}_{alpha}. Notice that P1 is trivially obeyed since there is only one DM. Since R^{i}_{alpha} precedes and conflicts with W^{j}_{alpha}, rule (2') implies that i must be serialized before j. Since R^{j}_{alpha} precedes and conflicts with W^{i}_{alpha}, the same rule implies that j must be serialized before i. Since both cannot be simultaneously satisfied, the execution is nonserializable. This occurred because the edges between $\bar{i}$ and $\bar{j}$ led to opposite serializations.

Protocol P3 prevents executions such as this one by making the following guarantee: If two transactions belong to two classes connected by a diagonal edge in a cycle, then the timestamp order of the two transactions is the same as

A Conflict Graph Cycle Figure 9.1

and Nonserializable Execution



Classes $\bar{I}$ and $\bar{J}$ have data item x in their read-sets and write-sets.

(a) The Conflict Graph

log for DM_{alpha}: R^{i}_{alpha} R^{j}_{alpha} W^{i}_{alpha} W^{j}_{alpha}

(b) A nonserializable log of transactions from class $\bar{I}$ and $\bar{J}$.

the relative ordering dictated by rules (1') or (2') applied to the messages that correspond to the edge. Before examining how P3 accomplishes this task, let us first see how P3 corrects the above example.

Since $[\langle r^{\bar{I}},w^{\bar{J}}\rangle,\langle w^{\bar{J}},r^{\bar{J}}\rangle,\langle r^{\bar{J}},w^{\bar{I}}\rangle,\langle w^{\bar{I}},r^{\bar{I}}\rangle]$ comprises a cycle, P3 applies to transactions i and j. Suppose that the timestamp of i is smaller than the timestamp of j. We observed that rule (2') required that i be serialized before j because R^i_{alpha} precedes W^j_{alpha}, and that j be serialized before i because R^j_{alpha} precedes W^i_{alpha}. But the latter requirement violates P3. Since $\langle r^j, w^i \rangle$ is in a cycle, protocol P3 implies that rule (2') applied to R^j_{alpha} and W^i_{alpha} must lead to i and j being serialized in timestamp order. However, the opposite occurred. What P3 must do, therefore, is make sure that W^i_{alpha} precedes R^j_{alpha}. Then both edges will lead to i and j being serialized in timestamp order and the nonserializability problem goes away.

Formally, we define protocol P3 as follows. *A transaction i obeys protocol P3 with respect to transaction j at DM_{alpha}* if R^i_{alpha} and W^j_{alpha} are processed in timestamp order. We require that for each diagonal edge $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ in a cycle and for each i, j and alpha such that R^i_{alpha} conflicts with W^j_{alpha}, i must obey P3 with respect to j at DM_{alpha}.

Protocol P3 synchronizes multi-class cycles as well as the simple two-class cycle just illustrated. In a cycle consisting of several diagonal and horizontal edges, P3 requires that each conflict due to a diagonal edge leads to the pair of transactions being serialized in timestamp order. Rule (3') makes the very same requirement for horizontal edges. So, insofar as this cycle is concerned, if rules (1')-(3') say anything about the relative ordering of two transactions whose classes are on the cycle, then the requirement must be that the transactions be serialized in timestamp order. Since there is only one timestamp ordering of transactions, conflicting serialization orderings are impossible. Generalizing this observation for the case of multiple transactions per class as we did for the acyclicity theorem leads to the correctness theorem for the SDD-1 concurrency control.

Serializability Theorem For a given set of transaction classes, $\underline{C}$, if

1. all classes in $\underline{C}$ obey the class pipelining rule, and
2. for each diagonal edge $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ in $CG_{\underline{C}}$ and transaction i in $\bar{I}$ and j in $\bar{J}$, transaction i obeys P1 with respect to transaction j, and
3. for each diagonal edge $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ in a cycle in $CG_{\underline{C}}$ and transaction i in $\bar{I}$ and j in $\bar{J}$, transaction i obeys P3 with respect to transaction j,

then all possible interleavings of transactions in classes in $\underline{C}$ are serializable.

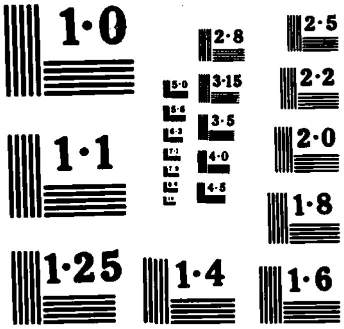
9.4 P2: A Faster Protocol for Read-Only Transactions

While P3 is sufficient for synchronizing all diagonal edges in a cycle, we can do somewhat better with those transactions that intersect the cycle only with their r-nodes. These read-only transactions contribute to nonserializability only because they may observe certain WRITE messages being processed in reverse timestamp order.* Protocol P2 is a weaker version of P3 that prevents this situation and thereby provides a less expensive alternative for synchronizing such transactions.

Suppose, for example, that the edges $\langle r^{\bar{i}},w^{\bar{j}}\rangle$ and $\langle r^{\bar{i}},w^{\bar{k}}\rangle$ appear in a conflict graph cycle. To synchronize a cycle, we want each set of transactions whose classes lie on the cycle to be serialized in timestamp order. If $\langle r^{\bar{i}},w^{\bar{j}}\rangle$ and $\langle r^{\bar{i}},w^{\bar{k}}\rangle$ are on this path, then transactions i, j, and k (say) in $\bar{i}$, $\bar{j}$, and $\bar{k}$ must be serialized in timestamp order. This two edge path will prevent a timestamp

* Strictly speaking, these transactions need not be read-only. It is just that their write operations, if they have any, do not participate in a conflict graph cycle.

AD-A155 724 A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND 2/4
AND CONTROL APPLICATIONS PART 2(U) COMPUTER CORP OF
AMERICA CAMBRIDGE MA 30 JAN 80 CCA-80-04-PT-2
UNCLASSIFIED N00039-77-C-0074 F/G 9/2 NL


1·0
2·8
2·5
5·0
3·15
2·2
5·6
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

ordered serialization only if transaction i observes WRITE messages from j and k in reverse timestamp order. For example, suppose TS_j and TS_k are the timestamps of j and k and $TS_j < TS_k$. If R^i_{alpha} precedes and conflicts with W^j_{alpha} and R^i_{beta} follows and conflicts with W^k_{beta}, then from i's viewpoint and according to rules (1') and (2'), k must be serialized before i which must be serialized before j. If either R^i_{alpha} had followed W^j_{alpha} or R^i_{beta} had preceded W^k_{beta}, j and k could have been serialized in timestamp order. Protocol P2 is designed to make precisely this guarantee.

A transaction i obeys protocol P2 with respect to transactions j and k if for any alpha

1. if R^i_{alpha} precedes and conflicts with W^j_{alpha} and $TS_k > TS_j$, then R^i_{beta} precedes W^k_{beta} at every DM_{beta} where they both appear and conflict, and

2. if R^i_{alpha} follows and conflicts with W^j_{alpha} and $TS_j > TS_k$, then R^i_{beta} follows W^k_{beta} at every DM_{beta} where they both appear and conflict.

That is, if $TS_j < TS_k$ then transaction i observes a WRITE message from transaction k only if it has observed all WRITE messages from transaction j, and conversely if $TS_k < TS_j$. Protocol P2 prevents i from observing a WRITE message from the later transaction unless it has observed all WRITE messages from the earlier one.

Protocol P2 is strictly weaker than P3 in that if i obeys P3 with respect to j and k then it obeys P2 with respect to j and k. Yet we can use it correctly for synchronizing classes which only intersect cycles with their r-nodes. Stated precisely, if $[\langle w^{\bar{j}}, r^{\bar{i}}\rangle, \langle r^{\bar{i}}, w^{\bar{k}}\rangle]$ is a subpath of a cycle, then if for each i, j, and k in $\bar{i}$, $\bar{j}$, and $\bar{k}$ we have that i obeys P2 with respect to j and k, then we need not synchronize these two diagonal edges using P3.

10. A Summary of the Protocol Selection Rules

In Section 9, we described the three basic protocols for synchronizing transactions and the conflict graph topologies that require the use of the protocols. While the analysis that leads to the protocols is somewhat complex, the rules for selecting the protocols are not. It is these Protocol Selection Rules that completely govern the concurrency control mechanism of SDD-1. We present these rules here in order to summarize and encapsulate the results of Section 9 and to incorporate a few more details to make the statement of the rules precise.

First, let us restate each of the three protocols.

Protocol P1: Transaction i obeys protocol P1 with respect to transaction j if for each DM, alpha, if W^{i}_{alpha} is processed before (resp. after) and conflicts with R^{j}_{alpha}, then W^{i}_{beta} is processed before (resp. after) R^{j}_{beta} at every DM_{beta} where they both appear and conflict.

Protocol P2: Transaction i obeys protocol P2 with respect to transactions j and k if for any alpha:

1. if R^{i}_{alpha} is processed before and conflicts with W^{j}_{alpha} and k has a later timestamp than j, then R^{i}_{beta} is processed before W^{k}_{beta} at every DM_{beta} where they both appear and conflict, and

2. if R^{i}_{alpha} is processed after and conflicts with W^{j}_{alpha} and j has a later timestamp than k, then R^{i}_{beta} is processed after W^{k}_{beta} at every DM_{beta} where they both appear and conflict,

Protocol P3: Transaction i obeys protocol P3 with respect to transaction j if for each DM_{alpha} at which R^{i}_{alpha} and W^{j}_{alpha} both appear and conflict, R^{i}_{alpha} and W^{j}_{alpha} are processed in timestamp order.

Briefly, these protocols serve the following purposes:

P1: Prevents READ messages from one transaction that conflict with WRITE messages from another transaction from being processed in different relative orders at different DM's.

P2: Prevents a READ message from seeing WRITE messages from two other transactions in reverse timestamp order.

P3: Prevents race conditions.

The Protocol selection rules state which protocols should be invoked by which transactions. They are:

I. For all classes in $\bar{I}$ and $\bar{J}$ such that $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ is in the conflict graph, for each pair of transactions i and j in $\bar{I}$ and $\bar{J}$ (respectively), i must obey protocol P1 with respect to j (see Figure 10.1a).

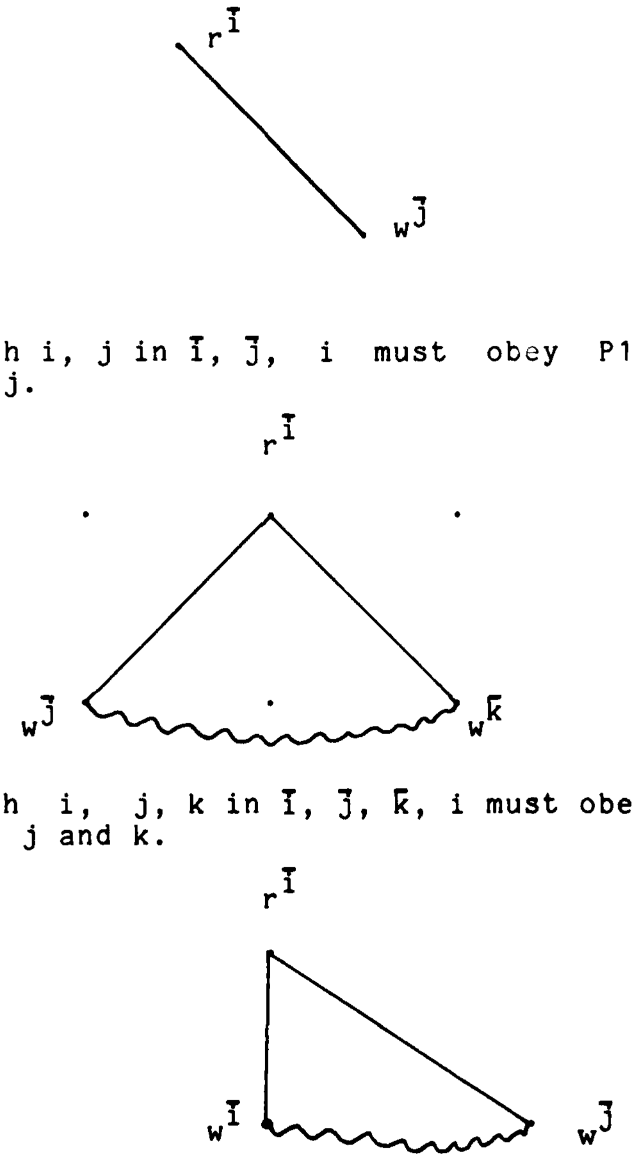
II. For each cycle in the conflict graph that contains a vertical edge, the following hold:

a. for all distinct classes $\bar{I}$, $\bar{J}$, $\bar{K}$, if edges $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ and $\langle r^{\bar{I}}, w^{\bar{K}} \rangle$ lie on the cycle, then for each set of transactions i, j, and k in $\bar{I}$, $\bar{J}$, and $\bar{K}$ (respectively), i must obey P2 with respect to j and k (see Figure 10.1a); and

b. for all distinct classes $\bar{I}$ and $\bar{J}$ such that $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ and $\langle r^{\bar{I}}, w^{\bar{J}} \rangle$ lie on the cycle, then for each pair of transactions i and j in $\bar{I}$ and $\bar{J}$ (respectively), i must obey P3 with respect to j (see Figure 10.1c).

The protocol selection rules are easily transformed into an algorithm that analyzes the conflict graph and produces the protocols that each class must obey. However, the definitions of the protocols are not algorithmic. To make the protocols effective, we now show how TMs and DMs can enforce the relative orderings of READ and WRITE messages required by the protocols.

Protocol Selection Rules Figure 10.1



(a) For each i, j in $\bar{I}$, $\bar{J}$, i must obey P1 with respect to j.

(b) For each i, j, k in $\bar{I}$, $\bar{J}$, $\bar{k}$, i must obey P2 with respect to j and k.

(c) For each i, i in $\bar{I}$, $\bar{J}$, j must obey P3 with respect to j.

11. Implementing the Protocols

11.1 Implementing Protocol P1

Each protocol demands that certain relative orderings of READ and WRITE messages be obeyed. Protocol P1 demands that READ and WRITE messages of two transactions that correspond to the endpoints of a diagonal edge must be processed in the same relative order at all DMs where they are both processed. Suppose the diagonal edge is $<r^{\bar{i}},w^{\bar{j}}>$. Then P1 says that if there are two DMs, alpha and beta, such that R^{i}_{alpha} and W^{j}_{alpha} are processed and conflict at DM_{alpha} and R^{i}_{beta} and W^{j}_{beta} are processed and conflict at DM_{beta}, then R^{i}_{alpha} is processed before W^{j}_{alpha} iff R^{i}_{beta} is processed before W^{j}_{beta}.

Let us first examine a simple case. If transactions in class $\bar{i}$ only send READ messages to one DM at which conflicting WRITE messages from class $\bar{j}$ are processed, then P1 is trivially satisfied. Since only one DM ever processes conflicting messages, there is no chance for a different ordering of conflicting messages at different DMs. If class $\bar{I}$ sends READ messages to two or more DMs at which conflicting WRITE messages from class $\bar{J}$ are processed, then synchronization is needed. The synchronization information is carried entirely by the READ messages from $\bar{I}$ in the form of read conditions.

A read condition is attached to a READ message and specifies which WRITE messages from certain other classes must be processed before the READ message can be correctly processed. The read condition includes a timestamp, say TS, and one or more classes, say $\{\bar{J}_1, \ldots, \bar{J}_m\}$. The read condition tells the DM to hold the READ message until such time that all WRITE messages from classes $\{\bar{J}_1, \ldots, \bar{J}_m\}$ with timestamps prior to TS have been processed and that no WRITE messages from classes $\{\bar{J}_1, \ldots, \bar{J}_m\}$ with timestamps later than TS have been processed. Then the READ message can be processed.

To implement protocol P1 on $\bar{I}$ with respect to $\bar{J}$, a read condition $\langle TS, \{\bar{J}\}\rangle$ must be attached to each READ message sent on behalf of a transaction i in $\bar{I}$ to each DM at which conflicting WRITE messages from $\bar{J}$ are processed. This is sufficient to guarantee P1. For example, if R^i_{alpha} is processed after W^j_{alpha}, then transaction j must have a timestamp prior to TS. So, at any other site, say beta,

R^{i}_{beta} will be processed after W^{j}_{beta} since the same read condition applies there as well. Notice that the choice of the timestamp TS is immaterial to the *correctness* of the protocol. All that matters is that all read conditions associated with i have the same timestamp. As we will see in a moment, the choice of timestamp can affect the *efficiency* of the protocol.

To correctly process a READ message with read condition <TS, $\{\bar{j}\}$> at a DM, the DM must wait until all WRITE messages from $\bar{j}$ with timestamps prior to TS have arrived and been processed. The class pipelining rule requires that WRITE messages from any given class be processed in timestamp order at every DM. So, as soon as the DM receives a WRITE message timestamped later than TS, it knows to hold it and process the READ message first. Of course, if a WRITE message from $\bar{j}$ with timestamp later than TS was processed before the read condition was received, then the READ condition cannot be satisfied without backing out the WRITE message. In SDD-1, no WRITE message is backed out for concurrency control reasons. So, in this case, the READ message would have to be *rejected* and the originating class must resubmit it with a later timestamp. Notice that *all* READ messages on behalf of transaction i have to be resubmitted, since their read conditions are now obsolete.

A problem with the mechanism described above is that class $\bar{j}$ may be idle because it has no transactions to process. The DM will therefore wait for a long time until a WRITE message timestamped later than TS arrives. One way to solve this problem is to have idle classes periodically send NULLWRITE messages.* A NULLWRITE message specifies the originating class and a timestamp and is interpreted as an empty WRITE message from that class with that timestamp. When a DM receives such a NULLWRITE message, it can be sure that it has received all WRITE messages from the indicated class through the given timestamp. If a DM chooses not to wait passively for a WRITE or NULLWRITE message from $\bar{j}$, it can request a NULLWRITE by sending a SENDNULL message to $\bar{j}$.

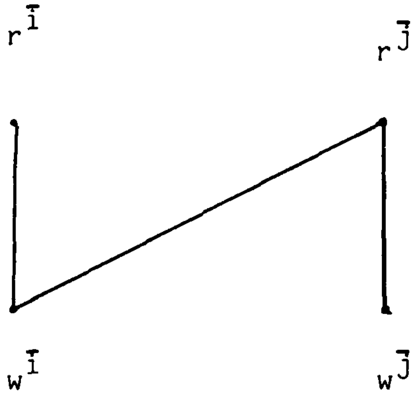
The choice of timestamps for read conditions and the rate at which NULLWRITEs are sent are important tuning parameters to avoid the frequent use of SENDNULLs. In addition, the choice of timestamp for read conditions will affect how long a READ message has to wait for conflicting WRITE messages to be processed. Essentially, the timestamp should be as small as possible without actually forcing the read condition to be rejected.

---

* The use of periodic NULLWRITE messages can be avoided by use of special protocols that are tailored for low

To illustrate the operation of protocol P1, let us consider a database that consists of two data items, x and y, where x is stored at DM_{alpha} and y is stored at DM_{beta}. Class $\bar{j}$ writes both x and y, and class $\bar{i}$ reads both x and y. For definiteness, suppose class $\bar{i}$ runs at $TM_{\bar{i}}$ and $\bar{j}$ runs at $TM_{\bar{j}}$. The conflict graph for this situation is shown in Figure 11.1. The edge $\langle r^{\bar{i}},w^{\bar{j}}\rangle$ implies transactions in $\bar{i}$ must obey P1 with respect to transactions in $\bar{j}$.

A Conflict Graph Illustrating P1 Figure 11.1



For $TM_{\bar{i}}$ to process a transaction, say i, it must send READ messages R^{i}_{alpha} to DM_{alpha} and R^{i}_{beta} to DM_{beta}. By P1, both messages must have a read condition $\langle TS,\{\bar{j}\}\rangle$ attached. DM_{alpha} will not process R^{i}_{alpha} to read x until

frequency classes. However, their description is beyond the scope of this paper.

it has received (but not processed) a WRITE message or a NULLWRITE from $TM_{\overline{j}}$ on behalf of $\overline{j}$ with timestamp later than TS. DM_{beta} will behave the same way. So, R^{i}_{alpha} will wait for (i.e., will be processed after) WRITE messages from the same set of transactions in $\overline{j}$ as R^{i}_{beta} will wait for. Hence, for each j in $\overline{j}$, rules (1') and (2') will require the same serialization order for i and j at both DM_{alpha} and DM_{beta}, and the result will be serializable. The nonserializable situation of R^{i}_{alpha} preceding W^{j}_{alpha} but R^{i}_{beta} following W^{j}_{beta} cannot occur.

11.2 Implementing P3

The same read condition mechanism that we described for implementing P1 is sufficient for implementing P3 as well. For transaction i to obey P3 with respect to all transactions j in $\overline{j}$ at DM_{alpha}, R^{i}_{alpha} must be processed after all W^{j}_{alpha} with earlier timestamps and before all W^{j}_{alpha} with later timestamps. If the timestamp of i is TS_{i}, then attaching the read condition $<TS_{i},\{\overline{j}\}>$ to R^{i}_{alpha} will force DM_{alpha} to process R^{i}_{alpha} according to P3; DM_{alpha} will wait for exactly those W^{j}_{alpha} with $TS_{j}<TS_{i}$.

From this implementation, we see immediately that protocol P3 is strictly stronger than protocol P1. If i obeys P3 with respect to j at DM_{alpha}, then i obeys P1 with respect to j at DM_{alpha}. The difference between P1 and P3 is that P1 allows any timestamp to appear in the read condition while P3 requires that timestamp to be TS_i.

Our earlier remarks about NULLWRITEs and SENDNULLs apply here as well. We noted under P1 that choosing a timestamp for the read condition was important to avoid lengthy delays. Since the read condition timestamp is the transaction's timestamp in P3, we must be careful to run the P3 transaction as early as possible -- early enough so that READ messages need not wait for many WRITE messages but not so early as to require its being rejected.

11.3 Implementing Protocol P2

As with the other protocols, P2 is implemented using read conditions. If i must obey P2 with respect to transactions in classes $\bar{j}$ and $\bar{k}$, then it must attach a read condition $<TS, \{\bar{j},\bar{k}\}>$ to each of its READ messages that are sent to a DM that processes conflicting WRITE messages from $\bar{j}$ or $\bar{k}$. As in P1, any timestamp for the read condition will do. Since some DMs will only process conflicting WRITE messages for either $\bar{j}$ or $\bar{k}$ (but not both) these DMs will only use one of the two classes in the second read condition parameter. If i conflicts with WRITE messages from $\bar{j}$ and $\bar{k}$ at only one DM, an interesting optimization is possible. Rather than specifying the timestamp TS in the read condition, the DM can select the timestamp itself. As long as there is some time, TS, such that all earlier WRITE messages and no later WRITE messages from $\bar{j}$ and $\bar{k}$ have been processed, P2 will be obeyed. However, if two or more DMs are involved, the timestamp must be fixed in advance, because all DMs must use the same timestamp; they cannot choose timestamps independently.

12. P4: A Cycle-breaking Protocol

Although P1, P2, and P3 are sufficient to guarantee serializability, from an efficiency standpoint these protocols have a very serious problem. The problem is that a single class can cause many cycles and thereby force many classes to use P2 and P3, even though very few transactions are ever run in that class.

While we expect that the vast majority of transactions that we wish to execute are predictable and belong to predefined classes, we still want to be able to execute an unexpected transaction that does not fit into any of our class definitions. One way to accomplish this is to define a very "large" class, call it $\bar{I}_{total}$, that has a read-set and write-set that includes the entire logical database. Every conceivable transaction can fit into $\bar{I}_{total}$, so this apparently solves the problem. But the cost is enormous, for $\bar{I}_{total}$ induces a two-class cycle with every other class in the system. So, every class has to run P3 against $\bar{I}_{total}$, and $\bar{I}_{total}$ has to run P3 against every other class. Since P3 is the most expensive protocol, this is an unfortunate state of affairs. It is especially unfortunate because transactions will rarely need to execute in $\bar{I}_{total}$, since most transactions fit into other less expensive classes. So, $\bar{I}_{total}$ introduces considerable synchronization overhead for synchronizing against a class that will rarely run a transaction.

In general, any class in which transactions are only infrequently run, but which creates many cycles in the conflict graph, exhibits this phenomenon. Although the problem of proliferation of cycles is especially acute in $\bar{I}_{total}$, other classes with smaller read-sets and write-sets may manifest the same problem.

To alleviate these problems we introduce a new protocol called P4, the purpose of which is to "break" cycles in the conflict graph. That is, if a class runs P4, then other classes that are in a cycle with the P4 class can behave as if the cycle did not exist.

One way to implement P4 is to shut off the system when a P4 transaction is introduced. No new transactions are processed and the syst?m works until all outstanding WRITE messages from transactions already in progress have been processed. When the system has finally quiesced, we can safely run the P4 transaction serially. After all of the P4 transaction's WRITE messages are processed, we can safely permit the system to process transactions again.

Since the execution before and after the P4 transaction ran was serializable (by the serializability theorem) and since the P4 transaction ran serially, the entire execution is serializable.

Of course, this implementation is likely to be unacceptable due to the severe performance degradation that results from shutting off the system, even temporarily. To improve the protocol, we first observe that a P4 transaction need only synchronize against classes that lie on a cycle that includes the P4 class, since only classes on cycles can cause nonserializability. Second, we note that even these classes need not quiesce completely before running a P4 transactions. Only conflicting WRITE messages must be completed before the P4 transaction executes and allows the other classes to resume processing. WRITE messages that do not conflict with READs in the same cycle cannot affect the ordering of transactions in the serialization according to rules (1')-(3'), and therefore they do not require synchronization under P4.

The implementation of P4 differs structurally from the other protocols in two ways. First, P4 requires some direct communication between TMs. By this communication, the P4 class requests that certain other TMs perform synchronization to avoid conflicting with the P4 transaction. Second, P4 requires an augmented form of read condition. Recall that a standard read condition is a pair of the form <timestamp, {classes}>. For P4, the timestamp may be interpreted as a "minimum time", i.e., <mintime=timestamp, {classes}>. This condition is satisfied if all WRITE messages from {classes} timestamped less than "timestamp" have been received. It does not require that no messages from {classes} timestamped greater than "timestamp" be received (as in standard read conditions).

To implement P4, we use three additional types of messages that are sent from TMs to TMs (not from TMs to DMs). A P4-ALERT message is sent from a P4 class to some other class. A P4-ALERT message includes the name of the P4 class and the timestamp of the P4 transaction as its parameters. A class responds to a P4-ALERT with either a P4-ACCEPT or a P4-REJECT.

To run a transaction i_{P4} in the P4 class $\overline{I}_{P4}$, one performs the following steps:

1. Choose a timestamp for i_{P4}, say TS_{P4}.

2. Send a message P4-ALERT (TS_{P4}) to every class that lies on the cycle with $\bar{i}_{P4}$ in the conflict graph.

3. Wait for the P4-ACCEPTs to be received from all classes to which a P4-ALERT was sent. If a P4-REJECT is received, then restart the protocol from step 1.

4. Construct the READ message for i_{P4}. For each DM_{alpha} and class $\bar{j}$ such that $<r^{\bar{i}_{P4}},w^{\bar{j}}>$ lies on a cycle and $\bar{j}$ sends WRITE messages to DM_{alpha} that can conflict with $R_{alpha}^{i_{P4}}$, attach the read condition $<TS_{P4}, \{\bar{j}\}>$ to $R_{alpha}^{i_{P4}}$.

When a TM receives a P4-ALERT (TS_{P4}) for a particular class, $\bar{j}$, it performs the following steps:

1. If $\bar{j}$ has run or begun running a transaction with a timestamp greater than TS_{P4}, then respond to $\bar{i}_{P4}$ by sending P4-REJECT. Otherwise, send P4-ACCEPT and do not run another transaction in $\bar{j}$ timestamped earlier than TS_{P4}.

2. In processing the next transaction run in $\overline{j}$, say j, for each DM_{alpha} to which j sends a READ message and for each class $\overline{k}$ such that $<r^{\overline{j}}, w^{\overline{k}}>$ lies on a cycle with $\overline{I}_{P4}$ and $\overline{k}$ sends WRITE messages to DM_{alpha}, attach the read condition $<mintime=TS_{P4}, \{\overline{k}\}>$ to R^{j}_{alpha}. These conditions are in addition to those normally carried by R^{j}_{alpha}. (Note: Only do this step for the <u>first</u> transaction in $\overline{j}$ with timestamp later than TS_{P4}.)

13. The Concurrency Monitor

The implementation of the run-time concurrency control mechanism primarily lies in a software module at the DMs called the Concurrency Monitor. The Concurrency Monitor at a DM accepts READ, WRITE, and NULLWRITE messages from TMs and schedules their execution at the DM. In essence, it is responsible for determining the ordering of events for local DM logs. In this section we will describe the operation of the Concurrency Monitor. As we will see, the mechanism is quite simple.

The Concurrency Monitor accepts and schedules messages of three types:

WRITE (TS, CLASS, UPDATES)

> TS is the timestamp of the transaction issuing the WRITE, and CLASS is its transaction class. UPDATES is a list of data item identifiers and values. When a WRITE is processed, the indicated data items are updated to the specified values according to the WRITE Message Rule (see Section 5.)

NULLWRITE (TS, CLASS)

> This message indicates that all future messages in CLASS will have timestamp greater than TS. Processing the NULLWRITE simply involves taking note of this fact in the internal tables of the Concurrency Monitor.

READ (TS, CLASS, READSET, CONDITIONS)

> TS and CLASS are the timestamp and transaction class of the transaction issuing the READ message. CONDITIONS is a list of read conditions associated with the READ message. Processing a READ involves reading the current values for data specified by READSET into a local transaction "workspace".

The read conditions have the following format:

<TYPE, CLASSES, TS>

> CLASSES is a list of transaction classes. TS is either a timestamp or is blank, depending on TYPE. If TYPE is "normal", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed, but no WRITE messages from those classes with greater timestamps have been processed. "Normal" read conditions are used in all four protocols. If TYPE is "DMchoice", then the TS specification is blank; the read condition is satisfied when the condition for "normal" read conditions can be satisfied for some selected value for TS. "DMchoice" read conditions are used in protocol P2. If type is "mintime", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed. "Mintime" read conditions are used in the P4 protocol. The TS specification in a read condition is always less than the transaction TS specified in the READ message itself.

The DM returns an ACCEPT-READ message when all the read conditions on a READ message have been satisfied and the READ has been processed. If the read conditions cannot be satisfied, even by waiting for new WRITE messages to be processed, then a REJECT-READ message is returned to the originator of the READ.

The function of the Concurrency Monitor is to schedule the processing of READ and WRITE messages under the constraints imposed by read conditions. READ messages can be processed as soon as they are satisfied. While WRITE messages should be processed without unnecessary delay, a WRITE message will be delayed if its immediate processing would cause the rejection of a pending READ message. When a READ message is received, it is checked to see if it is immediately rejectable. If it is not, then the READ will eventually be satisfied, because the Concurrency Monitor will not process any WRITE messages that will cause it to be rejected.

The Concurrency Table shown in Figure 13.1, contains the information needed by the Concurrency Monitor to resolve the status of read conditions. For each class, it holds a timestamp associated with the most recently processed WRITE or NULLWRITE message and a pointer to a queue of pending messages from that class to be processed. Within each queue, READ and WRITE messages appear in increasing timestamp order. This follows from the pipelining rules, and the fact that messages are guaranteed by the network to be received in the same order that they were sent. The Concurrency Monitor schedules the messages on each queue in the order that they appear. The message at the head of the queue is said to be <u>immediately pending</u>.

The Concurrency Table

Figure 13.1

Class	timestamp of most recently processed WRITE	timestamp of most recently processed NULLWRITE	pointer to pending message queue
$\bar{I}$	425179	425221	----------->
.	.	.	.
.	.	.	.
.	.	.	.

The Concurrency Monitor chooses the next message to be processed based on the following criteria:

1. Process any pending NULLWRITE.

2. *If there are none, process any immediately* pending WRITE, as long as this does not cause any pending READ to be rejected.

3. If there are no such WRITEs, process any immediately pending READs whose read conditions are satisfied.

It is important that the Concurrency Monitor not indefinitely postpone the processing of any immediately pending message either due to timing anomalies or deadlock. *One way to guarantee this would be to* schedule immediately pending messages according to the following priority rule. The priority of an immediately pending NULLWRITE or WRITE message is the TS parameter in the message; for a READ message, it is the lowest timestamp in an unsatisfied read condition in the READ. The Concurrency Monitor schedules smallest-priority-first.

We need to show that the smallest priority message can be processed within finite time. Let M be the message with lowest priority. If M is a NULLWRITE, it can be processed immediately. If M is a WRITE, then it will be held up only if there is an immediately pending READ with a read condition that has a timestamp smaller than M's. But then the READ would have a smaller priority than M, contradicting the choice of M. So, the WRITE can be immediately processed. Suppose that M is a READ. If it can be immediately processed then we are done. So, assume not and that $\langle TS_R, \{\bar{I},...\}\rangle$ is its unsatisfied read condition with lowest timestamp. Let M' be the immediately pending message on $\bar{I}$'s queue. If the queue is empty, a WRITE or NULLWRITE message with timestamp greater than TS_R will eventually appear on $\bar{I}$'s queue, since there are only a finite number of timestamps smaller than TS_R. If M' is a WRITE, then M' must have a timestamp greater than TS_R (by choice of M) and the READ condition is already satisfied, a contradiction. Similarly, M' cannot be a NULLWRITE. If M' is a READ, it must have an unsatisfied read condition $\langle TS'_R, \{\ldots\}\rangle$ with $TS'_R > TS_R$ (by choice of M). By pipelining rule C3, any WRITE message following M' has timestamp greater than M', and, as mentioned earlier, the timestamp of a transaction must be greater than the timestamp of its read condition. So, by transitivity, every WRITE message from $\bar{I}$ will have a timestamp greater than TS_R and again $\langle TS_R, \{\bar{I},\ldots\}\rangle$ is satisfied, a contradiction. Hence, the READ can be processed immediately. Since there are only a finite number of timestamps less than any priority, this also argues for proper termination, since every message will eventually be the one with the lowest priority.

It may not be wise to strictly follow this priority rule, since a lowest priority READ may wait for a while for all the necessary WRITEs to arrive. This would unnecessarily create a large backlog of other unprocessed messages. However, the above argument demonstrates feasibility; of course, any more efficient variation which never indefinitely postpones is also acceptable.

14. Reliability Considerations *

14.1 Overview

In this section we will describe the mechanisms by which the concurrency control algorithms are made resilient to failures of sites and communications facilities. These mechanisms provide two kinds of protection. First, the system must continue to operate correctly in the face of such failures. That is, the serializability guarantee must be maintained. Second, the procedures by which this is done must not force protocols to wait for failed sites to recover before they can safely proceed. Otherwise, transactions at non-failed sites could experience arbitrarily long delays before being allowed to run.

We will assume in this paper the existence of the Reliable Network (RelNet). The RelNet constitutes the

* The mechanisms reported in this section were developed by M. M. Hammer and D. W. Shipman.

communications component of the SDD-1 system. For our present purposes, it can be modelled as a virtual machine with the following properties:

1. The RelNet never fails.*

2. Between any sender-receiver pair, messages will be received in the order they were sent.

3. Messages are buffered within the RelNet. This implies that message delivery is guaranteed as soon as a message is accepted by the RelNet; the message need not arrive at its final destination for delivery to be assured. This means that messages sent to a failed site will be delivered upon its recovery.

4. The RelNet is aware of the updating activity of a transaction, and if the controlling TM fails before commiting a transaction, all of its updates will be aborted. No update becomes available for reading by another transaction until its transaction has been committed.

5. The RelNet maintains the clock used in assigning timestamps for concurrency control. The system behaves as if there were a single clock accessable to all sites.

6. The RelNet monitors site status. In addition to reporting status as up/down, the RelNet will indicate a clock interval during which that status applies.

---

* Of course, since 100% reliability is impossible to achieve, the actual RelNet may in fact fail. We consider this to be a "catastrophe" for which manual procedures may be required to repair any damage done.

A more detailed specification of the RelNet interface, as well as a description of its internal design, is given in [Hammer and Shipman]. Since the most difficult design issues have been relegated to the RelNet, all that is required here is to describe the ways these facilities are used in providing reliable and timely protocol implementations.

We need only be concerned with failures affecting the READ phase of a transaction. During the EXECUTE phase, the status of participating DMs is monitored by the TM, which will abort the transaction on a DM failure. Failures of the controlling TM during the EXECUTE phase result in transaction abortion by the RELNET. During the WRITE phase, if the controlling TM fails before all WRITE messages have been sent and the transaction committed, then the RELNET aborts the transaction, discarding any undelivered WRITEs. If the controlling TM fails after the transaction is committed, then all WRITEs are guaranteed to be safely delivered to their destinations by the RELNET.

We need to consider three issues arising in the READ phase:

1. the possibility that some data item in the read-set is not available.

2. the steps taken by the Concurrency Monitor when a read condition requires waiting for additional WRITE or NULLWRITE messages from a site which is down. Because the site may take arbitrarily long to recover, the Concurrency Monitor must be able to proceed in resolving the read condition without waiting for additional messages from that site.

3. the P4 protocol must be extended to deal with the situation in which an ACCEPT/REJECT response to a P4-ALERT message is required from a failed TM. Here again, it is unacceptable to wait for the failed site to recover in order for it to make the ACCEPT/REJECT decision.

The next three subsections deal with these issues.

14.2 Data Item Not Available

If all physical copies of a data item are unavailable because the DMs at which they are stored have failed, then the transaction cannot proceed. It is aborted and the user is informed.

It may happen that the originally chosen physical copy of the data item is unavailable, but that another copy of a data item is available at a different DM. In this case, the other copy is used for reading instead. It should be noted that the choice of which physical copies are to be read by a transaction does not affect the protocols which it must run. This is because the protocol requirements are expressed solely in terms of <u>logical</u> data item conflicts between transaction classes.

14.3 Read Conditions

When the timestamp on a read condition against a class is greater than the timestamp on any message which has been received from that class, it is necessary to wait until some message from that class arrives which has a greater timestamp than the read condition's. Only by waiting for such a message can the Concurrency Monitor be sure that it has knowledge of all WRITEs from that class with timestamps less than that specified in the read condition. If, however, the class in question runs at a TM which is down, it would seem that the Concurrency Monitor would have to wait for that TM to recover before the additional messages could be received.

The problem is solved as follows. Upon encountering a read condition which requires waiting for messages from a failed site, the Concurrency Monitor simply accepts the read condition. This is sound for the following reason. Upon recovery, all new transactions at the TM in question will have a timestamp greater than that of the read condition. This follows from the fact that the read condition timestamp is less than the timestamp of the transaction which issued it, that all transaction timestamps are obtained from the network clock and the fact that the network clock will have necessarily advanced past the timestamp of the reading transaction by the time the failed site recovers. Therefore it could not be possible for a WRITE message to arrive after the failed site's recovery which had timestamp less than that specified in the read condition, and it is thus safe to accept the read condition immediately.

14.4 Protocol P4

Protocol P4 calls for the issuing of a set of P4-ALERT messages to a number of TMs, and awaiting ACCEPT/REJECT responses. If a TM is down, it cannot, of course, respond and the P4 transaction would seem to have to wait for the TM's recovery.

Our solution to this problem is to assume an ACCEPT from any TM which was down at the time of the P4 transaction. Upon recovery, and before starting any transactions, the TM must read all messages which were sent to it while it was down (these have been buffered in the RelNet). If it finds a P4-ALERT in its message stream, it should process it as if it had been accepted. This approach is correct because: no transactions will have been processed at the recovering TM with timestamp greater than that of the P4 transaction (since the TM was down at the time of the P4); and all new transactions after the receipt of the P4-ALERT will have a timestamp greater than that of the P4 transaction. These are exactly the conditions necessary for acceptance of a P4-ALERT.

15. Advantages of the SDD-1 Concurrency Control Mechanism

The SDD-1 approach to concurrency control is in many ways quite different from other proposed mechanisms. We see many strengths in the approach. Unfortunately, there are few analytic methods for verifying these strengths, say by comparing the relative performance of our mechanism to other database concurrency controls. Furthermore, most of the proposed mechanisms are not yet implemented, so empirical comparisons are not possible either. Hence, the analysis of our mechanism must necessarily be more intuitive than mathematical. The specific criteria on which we base performance comparisons include: the amount of communication required to synchronize transactions; the average delay incurred by a transaction due to concurrency control; the amount of concurrency among transactions allowed by the concurrency control; and the overhead involved in making the mechanism resilient to communications and node failures.

At the architectural level, the SDD-1 concurrency control mechanism has two important properties. First, the architecture makes a strong separation between concurrency control issues and those of query processing and reliability. From a , oject management standpoint, this separation allowed us to attack the concurrecy control problem independently from and in parallel with query processing and reliability problems. From a software engineering standpoint, this division of labor led naturally to a division of function in software components. The concurrency control mechanisms are isolated in a small number of modules, making them easily modifiable and tunable.

Second, the architecture fully distributes the concurrency control. While each transaction is controlled from a single site, different sites are concurrently supervising the synchronization of many different transactions. No one site is in charge of <u>any</u> system-wide activity. The main advantage of this full distribution is enhanced reliability. A site failure only affects those transactions executing and/or using data at that site.

However, it is in the specific synchronization mechanisms that the most important advantages lie: conflict graph analysis and the timestamp-based protocols. We believe the technique of conflict graph analysis to be our most important contribution. By preanalyzing transaction conflicts, the number of transactions that need to be synchronized is drastically reduced. This has a beneficial effect on all aspects of concurrency control performance. It allows more concurrency among transactions; and for those transactions that require little or no synchronization it cuts delay, communications overhead, and costs associated with resiliency mechanisms. As shown in [BERNSTEIN and SHIPMAN b], the technique is quite general and can be used with a variety of synchronization protocols, including conventional locking. In principle, every proposed concurrency control mechanism could be improved by adding conflict graph analysis as a preprocessing step to eliminate run-time synchronization for some transactions.

The timestamp-based protocols, {P1,P2,P3,P4}, also offer important advantages over other proposed concurrency controls. First, the use of timestamps to resolve races among transactions eliminates the possibility of deadlock. Deadlock detection must be incorporated in any locking system and induces communications costs that the SDD-1 mechanism avoids. Second, the protocols synchronize transactions only against named transaction classes. Even if two transaction classes must be synchronized relative to certain data, other classes can concurrently access that data; in fact, other classes can independently be synchronized against that very same data without affecting the first two classes at all. This is in contrast to locking protocols, which set blanket locks that apply to all transactions that access the shared data. Third, SDD-1 offers a range of synchronization protocols. Protocol P2 is a fast synchronization protocol for read-only transactions that can afford to read an old, but consistent copy of the database. While with a locking strategy read-only transactions could choose not to lock the data they read, that unlocked data may be inconsistent. Protocol P4 allows infrequently executed transactions to take a larger share of the synchronization burden. By running such transactions under P4, other frequently executed transactions can run P1 with less delay and more concurrency than they would obtain if they ran P2 or P3 as otherwise required. The P4 capability is currently unique to the SDD-1 mechanism.

Quantitative comparisons among reliability mechanisms are not yet within the state-of-the-art. However, as indicated in the previous section, SDD-1 has incorporated recovery mechanisms that insulate it from the effects of network and node failure. The mechanisms are an example of a general approach to resiliency that we discuss in [HAMMER and SHIPMAN].

References

[ALSBERG and DAY]
Alsberg, P.A.; and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources", Report from the Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana Illinois, 1976.

[ALSBERG et al]
Alsberg, P.A.; Belford, G.G.; Bunch, S.R.; Day, J.D.; Grapa,E.; Healy, D.C.; McCauley, E.J.; and Willcox, D.A. Synchronization and Deadlock, CAC Document Number 185, CCTC-WAD Document Number 6503, Center for Advanced Computation, University of Illinois at Urbana, Illinois.-Champagne, Urbana

[BERNSTEIN and SHIPMAN a]
P. Bernstein and D. Shipman; "The Concurrency Control of SDD-1: A System for Distributed Databases; Part II: Analysis of Correctness"; submitted to Transactions on Database Systems.

[BERNSTEIN and SHIPMAN b]
Bernstein, P.A. and D.W. Shipman, "A Formal Model of Concurrency Control Mechanisms for Database Systems," Proc. 1978 Berkeley Workshop on Distributed Databases and Computer Networks.

[BERNSTEIN et al.]
Bernstein, P.A., Rothnie, J.B., Goodman, N., Papadimitriou, C.H.; "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans.on Soft. Eng., May 1978.

[CCA]
Computer Corporation of America, Datacomputer Version 5 User Manual, Cambridge, Massachusetts, July 1978.

[CHAMBERLIN et al]
Chamberlin, D.D.; Boyce, R. F.; Traiger, I.L. "A Deadlock-free Scheme for Resource Locking in a Database Environment", Information Processing 74, Proceedings AFIPS Conference, North Holland Publishing Company, Amsterdam, The Netherlands, 1974.

[ESWARAN et al]

Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[GRAY et al]

Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", Proc. Int'l Conf. on Very Large Data Bases, ACM, N.Y., Sept. 1975, pp. 428-451.

[HAMMER and SHIPMAN]

Hammer, M.M.; and Shipman, D.W., "Reliability Mechanisms in SDD-1: A System for Distributed Databases", submitted to the 1979 Transactions on Database Systems.

[KING and COLLMEYER]

King, P.F. and Collmeyer, A.J., "Database Sharing -- an efficient method for supporting concurrent processes", Proc. AFIPS 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 271-275.

[LAMPORT, L.]

"Time, Clocks and Ordering of Events in a Distributed System", Massachusetts Computer Associates Report #CA-7603-2911, March, 1976. Also submitted to CACM.

[MENASCE et al]

Menasce, D.A., G.J. Popek, and R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases", Proc. 1978 ACM-SIGMOD Conf. on Management of Data, ACM, N.Y.

[PAPADIMITRIOU et al]

Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[REED]

Reed, D.P., Naming and Synchronization in a Decentralized Computer System, Ph.D. Thesis, M.I.T., Sept. 1978.

[RIES and STONEBRAKER]
Ries, D.R. and M. Stonebraker, "Effects of Locking Granularity in a Database Management System", ACM Trans. on Database Systems, Vol. 2, No. 3 (Sept. 1977), pp. 233-246.

[ROSENKRANTZ et al]
Rosenkrantz, D.J.; Stearns, R.E.; and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems", ACM Trans. on Database Systems, Vol. 3, No. 2 (June 1978), pp. 178-198.

[ROTHNIE and GOODMAN]
Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.
(Also available from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-04.)

[ROTHNIE et al.]
Rothnie,J.B., Bernstein P.A., Fox, S., Goodman, N., Hammer, M.M., Landers, T., Reeve, C., Shipman, D.W., Wong, E.; "SDD-1: A System for Distributed Databases", submitted to the 1979 Transactions on Database Systems.

[STEARNS et al]
Stearns, R.E.; Lewis, P.M. II; and Rosenkrantz, D.J. "Concurrency Controls for Database Systems"; Proceedings of the 17th Annual Symposium on Foundations of Computer Science, IEEE, 1976, pp. 19-32.

[STONEBRAKER]
Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRESS", Proc. 3rd Berkeley Workshop in Distributed Data Management and Computer Networks, 1978, pp. 235-258.

[THOMAS a]
Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control", BBN Report 3340, July 1976.

[THOMAS b] Thomas, R.H., "A Solution to the Concurrency Control Problem for Multiple Copy Data Base", Proc. 1978 IEEE COMPCON Conf., IEEE, N.Y.

[WONG]
Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[WONG et al]
Wong, E., et al., "Query Processing in SDD-1: A System for Distributed Databases", submitted to the 1979 Transactions on Database Systems.

Concurrency Control in SDD-1: A System for Distributed Databases

Part II: Analysis of Correctness

Philip A. Bernstein
David W. Shipman

January 1, 1979

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Abstract

This paper presents a formal analysis of the concurrency control strategy of SDD-1. SDD-1, a System for Distributed Databases, is a prototype distributed database system being developed by CCA. In SDD-1, portions of data distributed throughout a network may be replicated at multiple sites. The SDD-1 concurrency control guarantees database consistency in the face of such distribution and replication.

Table of Contents

1. Introduction

SDD-1 is a prototype distributed database system being designed and implemented at Computer Corporation of America. The system involves a network of cooperating distributed processors behaving in an integrated fashion to provide users with a single consistent view of the complete database. The system allows for portions of the data to be stored redundantly at several network sites.

The concurrency control mechanism of SDD-1 ensures database and transaction consistency despite the interleaved nature of transaction execution in such an environment.

An informal description of the mechanism emphasizing implementation considerations appears in a companion paper, [BERNSTEIN et al.]. Our purpose in this paper is to prove formally the correctness of these mechanisms.

Our proof is entirely self-contained, but from a pedagogical standpoint it presupposes an intuitive understanding of the SDD-1 concurrency control mechanism. In this sense, it is a sequel to [BERNSTEIN et al.] which we strongly recommend be read before this paper. A review of other concurrency control mechanisms and a comparison of these other mechanisms also appears in [BERNSTEIN et al.].

2. A Formal Model of SDD-1

2.1 Introduction

To prove the correctness of the SDD-1 concurrency control mechanism, we need a formal model that describes the operation of an SDD-1 system. We describe such a formal model in this section. The model consists of a static component, called a database design, and a dynamic component, called an execution history. A database design describes the layout of data and of transaction classes in an SDD-1 system. An execution history describes the execution of transactions for a particular database design. Among the set of all possible execution histories, only some of these histories could be produced by the correct operation of the concurrency control mechanism. We call these histories well-behaved. Our goal, in Section 4, will be to show that all well-behaved histories produce correct results, i.e., that they are serializable.

2.2 Database Designs and Execution Histories

An *SDD-1 Database Design* consists of a set of data modules, which store data, and a set of classes, which can execute transactions.

Formally, it is a six-tuple <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset> where:*

i. DATA is a set of physical data items, denoted $\{x,y,z,\ldots\}$;

ii. DMs is a set of data modules, denoted {alpha, beta,...};

iii. stored-at:DATA -> DMs is a function that locates the data module that stores each data item;

iv. CLASSES is a set of transaction classes denoted $\{\bar{I},\bar{J},\bar{K},\ldots\}$;

v. c-readset:CLASSES -> 2^{DATA} defines the read-set of each class;

vi. c-writeset:CLASSES -> 2^{DATA} defines the write-set of each class.

* Notationally, upper case words are sets and lower case words are functions.

An SDD-1 Execution History is a representation of the execution of a set of transactions for a particular database design. Formally, it is a seven-tuple <$\underline{D}$, TRANS, classof, <, t-readset, t-writeset, LOGS> where:

i. $\underline{D}$ = <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset> is an SDD-1 database design;

ii. TRANS is a set of transactions, denoted {i,j,k,...};

iii. classof:TRANS -> CLASSES is a function denoting the class of each transaction (we will denote classof(i) as $\bar{I}$);

iv. < is a total order over TRANS. (In the SDD-1 implementation, i<j iff transaction i has a *smaller timestamp than transaction j*);

v. t-readset:TRANS -> DATA defines the read-set of each transaction such that for each i in TRANS, t-readset(i) is contained in c-readset(classof(i));

vi. t-writeset:TRANS -> DATA defines the write-set of each transaction such that for each i in TRANS, t-writeset(i) is contained in c-writeset(classof(i));

vii. LOGS is a set of ordered pairs $\text{LOGS}_{\text{alpha}}$ = < $\text{LOGELEMENTS}_{\text{alpha}}$, $=>_{\text{alpha}}$ > for all alpha in DMs such that

a. $\mathrm{LOGELEMENTS}_{alpha} = \mathrm{LOGELEMENTS}_{alpha,R} \cup \mathrm{LOGELEMENTS}_{alpha,W}$;

b. $\mathrm{LOGELEMENTS}_{alpha,R} = \{R^{i}_{alpha}$: i is in TRANS and there is an x in t-readset(i) such that stored-at(x) = alpha$\}$;

c. $\mathrm{LOGELEMENTS}_{alpha,W} = \{W^{i}_{alpha}$: i is in TRANS and there is an x in t-writeset(i) such that stored-at(x)=alpha$\}$;

d. $=>_{alpha}$ is a total order over $\mathrm{LOGELEMENTS}_{alpha}$ such that for all i in TRANS if R^{i}_{alpha} and W^{i}_{alpha} are in $\mathrm{LOGELEMENTS}_{alpha}$ then $R^{i}_{alpha} => W^{i}_{alpha}$.

The definition of an SDD-1 execution history is based on several facts about SDD-1's operation:

1. The total order < on transactions follows from the fact that transactions are given globally unique timestamps (i.e., (iv) above).

2. The read-set and write-set of a transaction must be a subset of the read-set and the write-set of that transaction's class (i.e., (v) and (vi) above).

3. Reads and writes are processed independently at different data modules. The order in which they are processed is modelled by =>. (Since the data module of interest is usually understood from the context, we normally drop the data module subscript on =>.)

4. If data item x is in transaction i's read-set, and x is stored at alpha, then R^{i}_{alpha} must appear in $LOGS_{alpha}$ (i.e., (vii.b) above).

5. If data item x is in transaction i's write-set, and x is stored at alpha, then W^{i}_{alpha} must appear in $LOGS_{alpha}$ (i.e.,(vii.c) above).

6. For each transaction, its read operation must precede its write operation at every DM where both are processed (i.e., (vii.d) above).

The total order < is used to determine how write operations affect the database. A write operation, say W^{i}_{alpha}, with x in t-writeset(i) actually writes a new value into x iff for all j with $W^{j}_{alpha} \Rightarrow W^{i}_{alpha}$ and x in t-writeset(j), j < i. That is, W^{i}_{alpha} writes into x iff it is "later" than all previous transactions that wrote into x at alpha. This mechanism, called the write message rule, is used to reorder write operations at each data module so that they appear to have occurred in "< -order" independent of their order of arrival [BERNSTEIN et al.].

2.3 Admissible Execution Histories

The concurrency control mechanism of SDD-1 consists of two components: a set of protocols, which are essentially procedures for processing a transaction's READ messages; and a set of protocol selection rules, which specify which protocols apply to transactions in each class (all transactions in a class use the same protocols). In a given execution history, if all transactions execute the protocols as specified by the protocol selection rules, then the execution history is called well-behaved. Intuitively, an execution history is well-behaved if all transactions follow the SDD-1 synchronization rules.

The main result of this paper is that all well-behaved execution histories are serializable. Since the motivation and formalism behind the SDD-1 concurrency control mechanism is based on some fundamental results about concurrency correctness in a database system, we must review these results before proceeding with a formal definition of well-behaved-ness and a proof of our theorem.

3. Serializable Histories

3.1 Consistency and Serializability

A prerequisite to proving the correctness of a system is a precise definition of what it means for the system to be correct. In SDD-1, we define the system to be correct if all possible histories are serializable. A serial history is one in which each transaction runs to completion before the next one starts. Thus, a serial history is one in which no concurrent activity has taken place. A serializable history is one which is equivalent to a serial history.

The intuitive justification for choosing serializability as the correctness criterion follows from the notion of consistency. Consistency may be considered to be a predicate over the state of the database; the database is either consistent or it is not. Each transaction submitted to the system is expected to preserve database consistency. That is, given a consistent database, it will always produce a consistent database. A serial history, therefore, necessarily preserves consistency if all the transactions involved preserve consistency. Since a serializable history is equivalent to a serial one, then it too preserves consistency. The use of serializability as a correctness criterion is nearly universal [ESWARAN et al.] [GRAY et al.] [HEWITT].

In order to define a serializable history as one which is equivalent to a serial history, we must be precise about what it means for two histories to be equivalent.

3.2 Equivalence

Intuitively, two histories are equivalent if they produce the same final state of the database for all interpretations of transactions and all initial database states. (To account for I/O, we assume that all input and output operations are treated as distinct data items.) Formalizing this concept of equivalence requires explicitly referencing initial and final database states. A neat way of incorporating the initial and final database states into the histories themselves is to augment each history with two dummy transactions called <u>first</u> and <u>last</u>. <u>first</u> initializes the database state at each data module:

1. for all i in TRANS (i ≠ first), first < i;

2. t-writeset (first) = DATA;

3. t-readset (first) = Ø; and

4. for each alpha in DMs and i in TRANS (i ≠ first), if R^{i}_{alpha} is in $LOGELEMENTS_{alpha}$ then W^{first}_{alpha} => R^{i}_{alpha}, and if W^{i}_{alpha} is in $LOGELEMENTS_{alpha}$ then W^{first}_{alpha} => W^{i}_{alpha} .

last reads the final database state at each DM:

1. for all i in TRANS (i ≠ last), i < last

2. t-readset (last) = DATA;

3. t-writeset (last) =Ø; and

4. for each alpha in DMs and i in TRANS (i ≠ last) if R^{i}_{alpha} is in $LOGELEMENTS_{alpha}$ then R^{i}_{alpha} => R^{last}_{alpha}, and if W^{i}_{alpha} is in $LOGELEMENTS_{alpha}$ then W^{i}_{alpha} => R^{last}_{alpha}.

To simplify notation in the sequel, we assume that all execution histories are augmented in this way.

The notion of equivalence of histories is characterized by the reads-from relation [PAPADIMITRIOU et al.]. Let H = <D, TRANS, classof, <, t-readset, t-writeset, LOGS> be an execution history. We say that R^i_{alpha} reads x from W^j_{alpha} in H iff

1. R^i_{alpha} and W^j_{alpha} are in $LOGELEMENTS_{alpha}$.
2. W^j_{alpha} => R^i_{alpha};
3. alpha = stored-at(x);
4. x is in t-readset(i);
5. x is in t-writeset(j); and
6. for all k in TRANS such that x is in t-writeset(k), if W^k_{alpha} => R^i_{alpha} then k < j.

Intuitively, if R^i_{alpha} reads x from W^j_{alpha}, then the value of x read by R^i_{alpha} is the value of x produced by W^j_{alpha}. Part (6) of the definition ensures that no other W^k_{alpha} produced the x-value read by R^i_{alpha} (cf. the write message rule at the end of Section 2.2). As a shorthand notation, if R^i_{alpha} reads x from W^j_{alpha}, then we also say that transaction i reads x from transaction j.

Not every read and write operation in H has an effect on the final database state produced by H. Those transactions that do have an effect are called live and are defined as follows:

1. last is live;

2. If transaction i is live, and for some x transaction i reads x from transaction j, then transaction j is live;

3. A transaction is live iff it so follows from (1) and (2).

A transaction that is not live is called dead.

Since every transaction at least prints something on an output device, no transaction is ever really dead. This can be modelled either by making each output operation write into a private data item, or by simply assuming that all transactions are live. We make the latter assumption for the remainder of this paper.

Two execution histories are equivalent if they have the same effect when applied to any database state. Formally, execution histories H_1 and H_2 are equivalent iff for every consistent database state S in domain(DATA) and for all interpretations of the transactions, H_1 and H_2 map S into the same final state, S_f. The following lemma characterizes equivalence of execution histories.

Lemma E [PAPADIMITRIOU et al. 77]

Two histories, H_1 and H_2 are equivalent iff $TRANS_1$ and $TRANS_2$ have the same set of live transactions and for all live i,j in $TRANS_1$, if transaction i reads some data item x from transaction j in H_1 then transaction i reads x from transaction j in H_2.

This is a standard program schema theoretic result, and can be found applied to a variety of models (e.g., [MANNA]). It can intuitively be justified by observing that if a transaction reads the same input data in both histories, then it will perform the same computation in both histories. The condition of Lemma E guarantees that each transaction reads the same inputs in both histories, thereby guaranteeing equivalence. The converse follows from the fact that the equivalence must hold over all interpretations of the transactions.

3.3 Serializability

Let {A,B,C,D} denote log symbols each of which is either an R or a W. Using this notation, we now define the LOGS of history H to be <u>serial</u> if there is no A^i_{alpha}, B^j_{alpha}, C^i_{beta}, D^j_{beta}, such that i≠j, A^i_{alpha} => B^j_{alpha} and D^j_{beta} => C^i_{beta}. That is, a serial log is one in which no two transactions are interleaved; transactions execute in the same order at all data modules. An execution history is <u>serial</u> iff its LOGS is serial. As discussed earlier, serial execution histories are our benchmark for consistent executions.

An execution history is *serializable* if it is equivalent to some serial execution history. Since serial execution histories preserve database consistency, serializable execution histories preserve database consistency as well.

Given an execution history, H, we can determine whether or not H is serializable by defining a graph on H called a serialization graph. The *serialization graph* on history H, denoted SG(H), is a node-labelled directed graph <V,E> where:

$V = \{i \mid \text{all } i \text{ in TRANS}\};$

$E = E_{\text{reads-from}} + E_{\text{interferes}};$

$E_{\text{reads-from}} = \{\langle i,j \rangle \mid j$ is live and j reads some data item x from $i\}$

$E_{\text{interferes}} = \{\langle k,i \rangle \mid$ for some j in TRANS, j is live, j reads some data item x from i, x is in t-writeset(k), and $k < i\} +$
$\{\langle j,k \rangle \mid$ for some i in TRANS, j reads some data item x from i, x is in t-writeset(k), and $k > i\};$

where '+' denotes set union.

The edges in a serialization graph reflect the notion of "happened before." The edges in $E_{interferes}$ guarantee that if j reads x from i in H, then in the serialization of H no k that writes into x appears in a position that would have j read x from k (instead of from i).

Theorem SER If SG(H) is acyclic then H is serializable.

Proof

Since SG(H) is acyclic, we can topologically sort its nodes, i.e. the elements of TRANS. This topological sort induces a serial log, say LOGS', on the transactions in TRANS, i.e., each $LOGS_{alpha}$ in LOGS' uses the sequence of serial transactions specified by the topological sort. Let H' be a history that differs from H only in that H' uses LOGS' instead of LOGS. If H' is equivalent to H, then H is serializable.

To show that H' is equivalent to H, we must show that for all i,j in TRANS, if j reads some x from i in H then j reads x from i in H' (by Lemma E). Suppose j reads x from i in H. Then <i,j> is in $E_{reads-from}$ in SG(H) (denoted $E_{reads-from}(H)$), so i precedes j in LOGS'. By $E_{interferes}(H)$, for all k that also write into x, if k > i then j precedes k in LOGS'. So j reads x from i in H' as well. (Notice that this is true whether or not the write message rule is used in H'.) Q.E.D.

An iff version of version of Theorem SER appears in [PAPADIMITRIOU et al.] using a more general definition of serialization graph, but without the <-ordering of transactions. The latter forced us to reprove the theorem here.

Corollary SER If graph G contains all the edges of serialization graph SG(H), and G is acyclic, then H is serializable.

4. Well-Behaved Execution Histories

4.1 Time-ordered Serialization Graphs

We define a relation over transactions in an execution history, H, called the time-ordered serialization graph, denoted by ->. Intuitively, i->j implies transaction i must precede transaction j in any serialization. We will show that the -> relation is a superset of SG(H).

For a given H, we define -> as follows:

1. $\rightarrow \; = \; \rightarrow_{rw} + \rightarrow_{wr} + \rightarrow_{ww}$

2. $i \rightarrow_{rw} j$ iff $i \neq j$ and there exists alpha in DMs such that $R^{i}_{alpha} \Rightarrow W^{j}_{alpha}$ and there exists some x such that stored-at(x) = alpha, x is in t-readset(i) and x is in t-writeset(j).

3. $i \rightarrow_{wr} j$ iff $i \neq j$ and there exists alpha in DMs such that $W^{i}_{alpha} \Rightarrow R^{j}_{alpha}$ and there exists some x such that stored-at(x) = alpha, x is in t-writeset(i), and x is in t-readset(j).

4. $i \rightarrow_{ww} j$ iff $i<j$ and the intersection of t-writeset(i) and t-writeset(j) is non-empty.

The relation -> contains the serialization graph for H. That is, the graph $SG_0(H)$ defined as follows:

V_{SG_0} = {i¦ transaction i appears in H};
E_{SG_0} = {<i,j>¦ i -> j};

contains the serialization graph for H.

Lemma TOSG $SG_0(H)$ contains the serialization graph for H.

Proof

We must show that $E_{reads\text{-}from}$ and $E_{interferes}$ are included in the edge set of $SG_0(H)$.

To show that $E_{reads\text{-}from}$ is contained in $E_{SG_0(H)}$, we simply note that $i \rightarrow_{wr} j$ subsumes $E_{reads\text{-}from}$.

To show that $E_{interferes}$ is contained in $E_{SG_0(H)}$, suppose R^{j}_{alpha} reads x from W^{i}_{alpha} in H, and consider some other W^{k}_{alpha} that also writes into x. So, W^{k}_{alpha} and W^{i}_{alpha} intersect in x. If $k < i$, then $k \rightarrow_{ww} i$, so <k,i> is in

$E_{SG_0(H)}$ as desired. So, suppose i < k. Since R^j_{alpha} reads x from W^i_{alpha}, it follows that $R^j_{alpha} \Rightarrow W^k_{alpha}$. (If $W^k_{alpha} \Rightarrow R^j_{alpha}$, then R^j_{alpha} reads x from W^k_{alpha} rather than W^i_{alpha}, independent of the relative ordering of W^i_{alpha} and W^k_{alpha}.) Thus, $j \rightarrow_{rw} k$, so <j,k> is in $E_{SG_0(H)}$ as desired. Hence, $E_{interferes}$ is contained in $E_{SG_0(H)}$. Q.E.D.

Corollary TOSG If $SG_0(H)$ is acyclic, then H is serializable.

Proof Follows directly from lemma TOSG and corollary SER. Q.E.D.

Execution histories, as defined in Section 2, may produce cyclic serialization graphs. However, if an execution history satisfies the SDD-1 synchronization rules, then cycles are not possible. In Section 4.2 we precisely define those execution histories that satisfy the SDD-1 synchronization rules. In Section 4.3 we show that all such histories produce acyclic time-ordered serialization graphs. Combined with corollary TOSG, this is sufficient to show that such histories are serializable.

4.2 The SDD-1 Synchronization Mechanism

Intuitively, an execution history is "well-behaved" if each transaction in the history obeys the required protocols. The required protocols are determined by analyzing the database design, thereby assigning a set of protocols to each class. Each transaction is required to satisfy all of the protocols assigned to some class of which it is a member. To formalize these concepts, we first explain what transactions must do to satisfy the protocols and then describe how protocols are assigned to classes of transactions.

There are four basic protocols in SDD-1: P1, P2, P3, and P4. A protocol is a property that an execution history must satisfy; it is implemented as an algorithm for executing read messages on behalf of transactions, which thereby only allows well-behaved histories to be produced [BERNSTEIN et al.].

Let H be an execution history. We say that H obeys protocol P1 with respect to classes $\bar{I}$ and $\bar{J}$ (written P1($\bar{I}$,$\bar{J}$)) iff for each pair of transactions i and i' in $\bar{I}$ and j and j' in $\bar{J}$, (P1) $j \rightarrow_{wr} i \leq i' \rightarrow_{rw} j'$ implies $j < j'$.

(Note: $i \leq i'$ means that either $i < i'$ or i is identical to i'.)

H obeys protocol P2 with respect to class $\bar{I}$ and a set of classes $j = \{\bar{J}^1, \ldots, \bar{J}_n\}$ (written P2($\bar{I}$,J)) iff for each pair of transactions i and i' in $\bar{I}$ and j in $\bar{J}_u$ and j' in $\bar{J}_v$ (for some $\bar{J}_u$ and $\bar{J}_v$ in J)

(P2) $j \rightarrow_{wr} i \leq i' \rightarrow_{rw} j'$ implies $j < j'$.

H obeys protocol P3 with respect to classes $\bar{I}$ and $\bar{J}$ (written P3($\bar{I}$, $\bar{J}$)) iff for each transaction i in $\bar{I}$ and j in $\bar{J}$,

(written P3($\bar{I}$, $\bar{J}$)) iff for each transaction i in $\bar{I}$ and j in $\bar{J}$,

(P3) $i \rightarrow_{rw} j$ implies $i < j$
and $j \rightarrow_{wr} i$ implies $j < i$.

H obeys protocol P4 with respect to class $\bar{I}$ and a set of classes $j = \{\bar{J}^1, \ldots, \bar{J}_p\}$ (written P4($\bar{I}$,J))

(written P4($\bar{I}$, J)) iff for each transaction i in $\bar{I}$, j in $\bar{J}_u$, and j' in $\bar{J}_v$ (for some $\bar{J}_u$ and $\bar{J}_v$ in J),

(P4) $j \rightarrow j'$ and $i \leq j$ imply $i < j'$
and $j' \rightarrow j$ and $j \leq i$ imply $j' < i$.

The motivation for the protocols can be found in [BERNSTEIN et al.]. To review the uses of the protocols:

- P1 incorporates class pipelining and synchronizes simple read-write conflicts;
- P2 avoids having a read operation see updates in reverse timestamp order;
- P3 avoids update race conditions;
- P4 is the "cycle-breaking protocol" for unanticipated and very infrequent transactions.

Each transaction executes in a class. The protocols that each class must obey are determined from an analysis of the database design, using a mathematical structure called a conflict graph.

A <u>conflict graph</u> for a database design D = <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset>, denoted CG(D), is an undirected node-labelled graph <V,E> where

$V = \{r^{\bar{I}} \mid \bar{I}$ in CLASSES$\} + \{w^{\bar{I}} \mid \bar{I}$ in CLASSES$\}$;

$E = E_{vert} + E_{horiz} + E_{diag}$;

$E_{vert} = \{\langle r^{\bar{I}}, w^{\bar{I}}\rangle \mid \bar{I}$ in CLASSES$\}$;

$E_{horiz} = \{\langle w^{\bar{I}}, w^{\bar{J}}\rangle \mid \bar{I}, \bar{J}$ in CLASSES and the intersection of c-writeset$(\bar{I})$ and c-writeset$(\bar{J})$ is nonempty$\}$

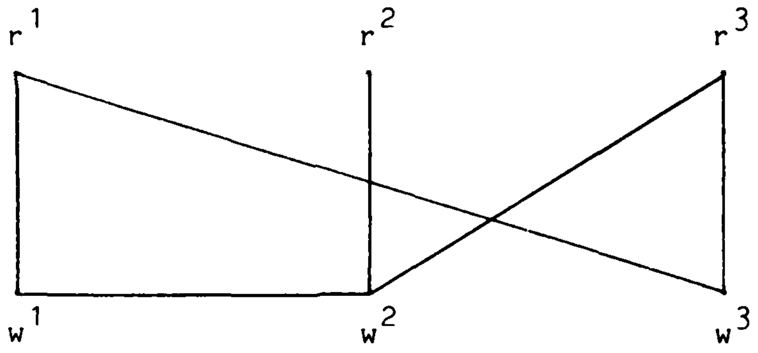
$E_{diag} = \{\langle r^{\bar{I}}, w^{\bar{J}}\rangle \mid \bar{I}, \bar{J}$ in CLASSES and the intersection of c-readset$(\bar{I})$ and c-writeset$(\bar{J})$ is nonempty$\}$

By convention, for each $\bar{I}$ the node $r^{\bar{I}}$ is drawn above $w^{\bar{I}}$ and for each $\bar{I}$ and $\bar{J}$ the nodes $r^{\bar{I}}$ and $r^{\bar{J}}$ are colinear on a horizontal line as are the nodes $w^{\bar{I}}$ and $w^{\bar{J}}$. This leads to the concepts of vertical, horizontal and diagonal edges (see Figure 4.1).

A Class Conflict Graph

Figure 4.1

Class:	1	2	3
Readset:	{a}	{c}	{d}
Writeset:	{b}	{b,d}	{a}



A *path* in a conflict graph is a sequence of edges $[<i_0, i_1>, <i_1, i_2>, <i_2, i_3>,\ldots,<i_{n-1}, i_n>]$ taken from E. A *cycle* is a path in which $i_0 = i_n$. We call edges in E_{horiz} and E_{diag} *heterogeneous*, since they are incident with two distinct classes. A cycle is *nonredundant* if no more than two heterogeneous edges in the cycle are incident with any class (whether the class nodes are r's, w's or include one of each is not significant with respect to redundancy).

An execution history H on database design D is called *well-behaved* if it satisfies the following *protocol selection rules*:

I. For each nonredundant cycle, nrc, in CG(D) that contains a vertical edge (i.e., $\langle r^{\bar{I}}, w^{\bar{I}}\rangle$), either

1. for some class $\bar{I}$ in the set of classes, J, that are incident with nrc, H obeys P4($\bar{I}$, J); or

2. each of the following hold (see Figure 4.2):

   a. for all classes $\bar{I}$, $\bar{J}$, $\bar{K}$ such that $\bar{I} \neq \bar{J} \neq \bar{K}$, if edges $\langle r^{\bar{I}}, w^{\bar{J}}\rangle$ and $\langle r^{\bar{I}}, w^{\bar{K}}\rangle$ are in nrc, then H obeys P2($\bar{I}$,{$\bar{J}$, $\bar{K}$}); and

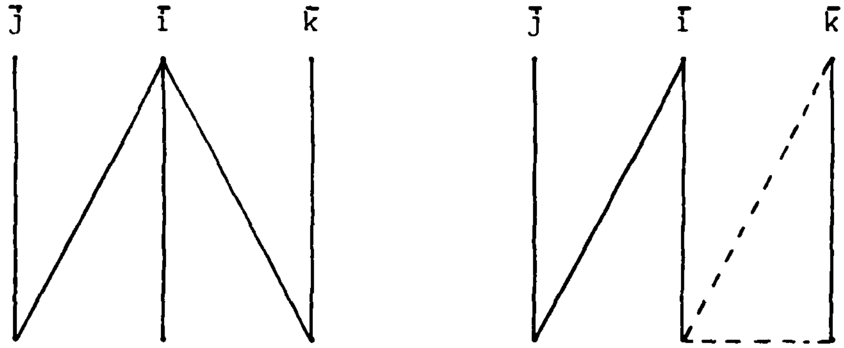
   b. for all classes $\bar{I}$, $\bar{J}$ such that $\langle r^{\bar{I}}, w^{\bar{J}}\rangle$ and either $\langle w^{\bar{I}}, w^{\bar{K}}\rangle$ or $\langle w^{\bar{I}}, r^{\bar{K}}\rangle$ (for some $\bar{K}$) are in nrc, then H obeys P3($\bar{I}$, $\bar{J}$).

II. For all classes $\bar{I}$, $\bar{J}$ such that $\langle r^{\bar{I}}, w^{\bar{J}}\rangle$ is in CG(D), H obeys P1($\bar{I}$, $\bar{J}$).

III. For all classes, $\bar{I}$, such that readset($\bar{I}$) intersects writeset($\bar{I}$), H obeys P3($\bar{I}$, $\bar{I}$).

Protocol Selection Rule I-2

Figure 4.2



Case I-2a

Case I-2b

4.3 Proof of Serializability

In corollary TOSG, we showed that for a given execution history, H, if $SG_0(H)$ is acyclic, then H is serializable. We now complete the proof of serializability by showing that if H is well-behaved then $SG_0(H)$ is acyclic.

<u>Lemma ACYC</u> If execution history H is well-behaved, then $SG_0(H)$ is acyclic.

To prove this lemma, we need to show that the existence of a cycle in $SG_0(H)$ leads to a contradiction. As a preliminary step, we first examine special edge sequences in $SG_0(H)$, called trails. We will show that the endpoints of a trail, i and j say, are in timestamp order, i.e. $i < j$ (see lemma TRAIL and lemma TRAIL-P4 below). Then, given any cycle in $SG_0(H)$, we show that for some transaction i in the cycle, there is a trail from i to i. But, by the previous result, this implies $i < i$, a contradiction. Hence, the cycle cannot exist. We proceed formally.

We define a trail (see Figure 4.3) in $SG_0(H)$ to be a sequence of n edges $<i_0, i_1>$, $<i_2, i_3>$,..., $<i_{2n-2}, i_{2n-1}>$ such that

1. n is greater than 0;

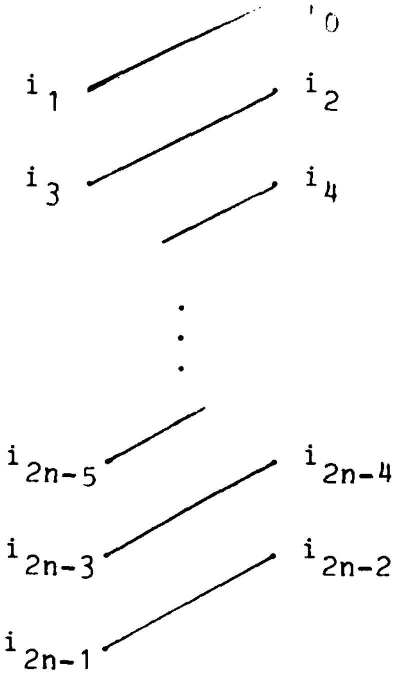
2. for j between 1 and n-1, $\text{classof}(i_{2j-1}) = \text{classof}(i_{2j})$ and $i_{2j-1} \leq i_{2j}$;

3. for j and k between 0 and n-1, with $j \neq k$, $\text{classof}(i_{2j}) \neq \text{classof}(i_{2k})$.

Lemma TRAIL Let H be a well-behaved execution history and let -> and $SG_0(H)$ be defined on H. Let $T = [<i_0, i_1>, <i_2, i_3>,...,<i_{2n-2}, i_{2n-1}>]$ be a trail in $SG_0(H)$ where $\text{classof}(i_0) = \text{classof}(i_{2n-1})$ and no transaction in the trail ran P4 with respect to the other classes in the trail. Then $i_0 < i_{2n-1}$.

A Trail

Figure 4.3



a. transactions on the same horizontal line are in the same class (e.g., $\bar{I}_1 = \bar{I}_2$). Furthermore, the transaction on the left $\le$ the transaction on the right. (e.g., $i_1 \le i_2$).

b. transactions on different horizontal lines are in different classes (e.g., $\bar{I}_1 \neq \bar{I}_4$).

Proof

Every edge in the trail corresponds to an edge in CG(D) in the following sense: for each j between 0 and n-1,

1. $i_{2j} \rightarrow_{wr} i_{2j+1}$ implies $<w^{\bar{I}_{2j}}, r^{\bar{I}_{2j+1}}>$ is in CG(D);

2. $i_{2j} \rightarrow_{rw} i_{2j+1}$ implies $<r^{\bar{I}_{2j}}, w^{\bar{I}_{2j+1}}>$ is in CG(D); and

3. $i_{2j} \rightarrow_{ww} i_{2j+1}$ implies $\langle w^{i_{2j}}, w^{i_{2j+1}} \rangle$ is in CG(D).

So, trail T corresponds to a sequence of edges, T_{CG}, in CG(D). We now prove two facts about T_{CG}.

Claim 1: The only cases in which T_{CG} contains two identical edges are when n = 2 and T_{CG} is of the form $[\langle r^i, w^j \rangle, \langle w^j, r^i \rangle]$, $[\langle w^i, r^j \rangle, \langle r^j, w^i \rangle]$, or $[\langle w^i, w^j \rangle, \langle w^j, w^i \rangle]$ where $i \neq j$.

Proof of claim 1: If n = 2, then the above forms are the only ones possible, such that T satisfies the definition of trail and T produces two identical edges in T_{CG}. So, suppose T contains more than two edges, and two of its edges, say $\langle u, v \rangle$ and $\langle x, y \rangle$, produce identical edges in T_{CG}. By construction, all edges in T are "heterogeneous" (i.e., are incident with distinct classes), and therefore produce heterogeneous edges in T_{CG}. So, either the pairs u-x and v-y are each incident with the same class, or the pairs u-y and v-x are each incident with the same class. As long as T contains at least three edges, this implies that part(3) of the definition of trail is violated. (For example, if $\langle u, v \rangle$ and $\langle x, y \rangle$ are adjacent, then the head of the edge preceding $\langle u, v \rangle$ is in the same class as y, and the tail of the edge following $\langle x, y \rangle$ is in the same class as u, thereby violating part (3) of the definition. Other cases follow by the same argument.)

Claim 2: For n greater than 2, T_{CG} can be augmented by homogeneous (i.e., vertical) edges to create a nonredundant cycle T'_{CG}.

Proof of claim 2: The head and tail of adjacent edges in T_{CG} are in the same class (by part(2) of the definition of trail). If they are not identical nodes, then they can be connected by a vertical edge (since they must be the r and w node of a single class). Insert all such vertical edges, creating a path T'_{CG}. No vertical edge in CG(D) will be added more than once (because of part (3) of the definition of trail). This fact, combined with claim 1, shows that T'_{CG} is a cycle. Part (3) of the definition of trail demonstrates that T'_{CG} is nonredundant.

Having set the stage with claim 2, we now proceed to prove the lemma by showing it to be true in each of the following three casees:

I. n = 1;

II. n = 2;

III. n greater than 2.

The cases subsume all possible trails T and therefore are sufficient to prove the lemma.

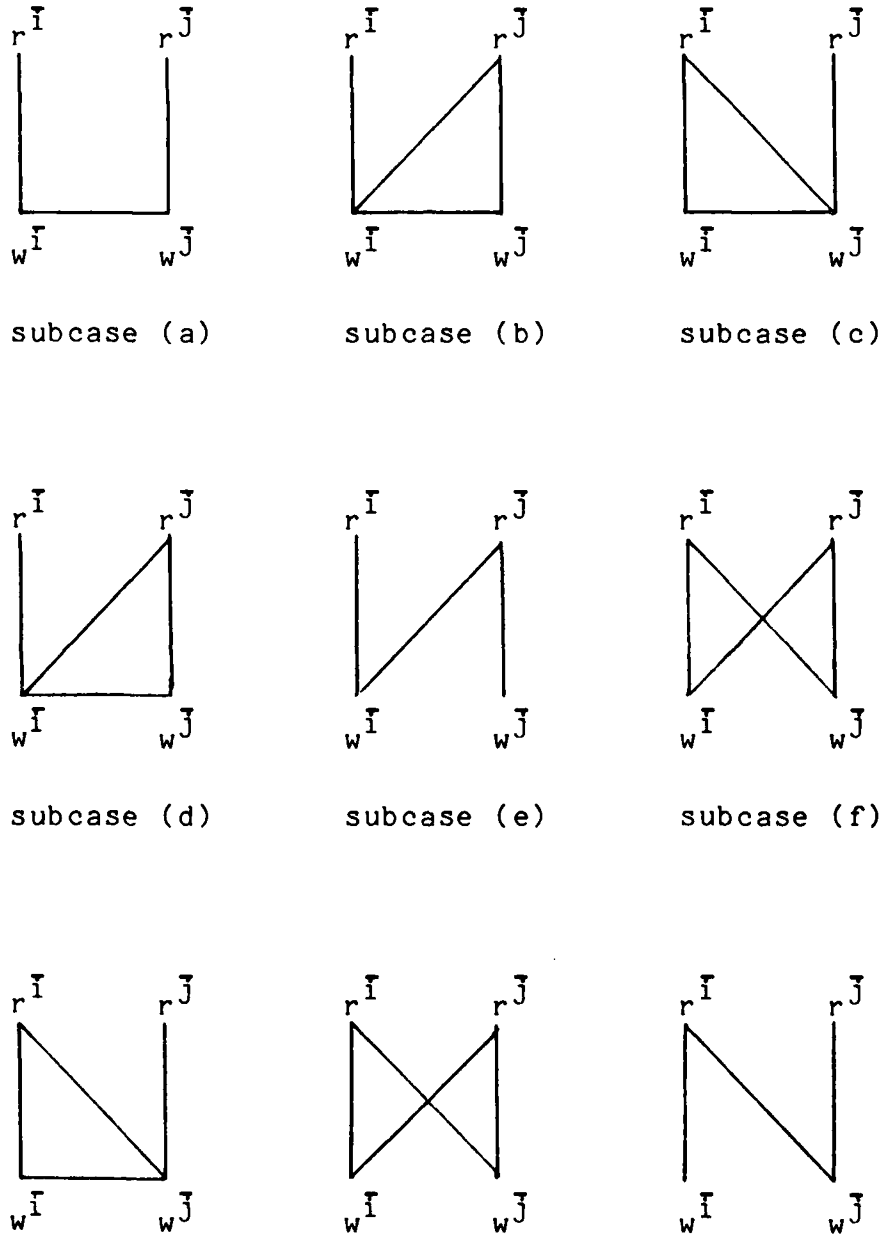
<u>Case I</u> Assume n = 1. Then T must be of the form [<i_0, i_1>]. Either $i_0 \rightarrow_{ww} i_1$ or $i_0 \rightarrow_{rw} i_1$ or $i_0 \rightarrow_{wr} i_1$. If $i_0 \rightarrow_{ww} i_1$, then $i_0 < i_1$ by definition of $\rightarrow_{ww}$. If $i_0 \rightarrow_{rw} i_1$ then t-readset(i_0) intersects t-writeset(i_1). So, the readset and writeset of $\bar{I}$ = classof(i_0) = classof(i_1) intersect. By protocol selection rule III, P3($\bar{I}$, $\bar{I}$) must be obeyed. Hence, $i_0 \rightarrow_{rw} i_1$ implies $i_0 < i_1$. If $i_0 \rightarrow_{wr} i_1$, then $i_0 < i_1$ follows by the same argument.

<u>Case II</u>: Assume n = 2. Then T must be of the form [<i_0, j_0>, <j_1, i_1>]. Let $\bar{I}$ = classof(i_0) = classof(i_1) and $\bar{J}$ = classof(j_0) = classof(j_1). There are nine subcases to consider, depending on the way the i's and j's are related by -> (see Figure 4.4). Note that $j_0 \leq j_1$ by definition of trail.

In the first three subcases, we assume $i_0 \rightarrow_{ww} j_0$. By definition of $\rightarrow_{ww}$, we have $i_0 < j_0$. Since $j_0 \leq j_1$, by transitivity $i_0 < j_1$. In each subcase, we will show $j_1 < i_1$, thereby showing that $i_0 < i_1$.

<u>Subcase a</u>: Suppose $j_1 \rightarrow_{ww} i_1$. By definition of $\rightarrow_{ww}$, $j_1 < i_1$. So, by transitivity $i_0 < i_1$.

Conflict Graphs for Case II of Lemma Trail Figure 4.4



subcase (a) subcase (b) subcase (c)

subcase (d) subcase (e) subcase (f)

subcase (g) subcase (h) subcase (i)

Subcase b: Suppose $j_1 \text{->}_{rw} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{I}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, r^{\bar{J}}\rangle, \langle r^{\bar{J}}, w^{\bar{I}}\rangle]$. By protocol selection rule I.2b, P3($\bar{J}$, $\bar{I}$) must be obeyed. Hence, $j_1 \text{->}_{rw} i_1$ implies $j_1 < i_1$, and by transitivity $i_0 < j_1$.

Subcase c: Suppose $j_1 \text{->}_{wr} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{I}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, r^{\bar{I}}\rangle, \langle r^{\bar{I}}, w^{\bar{I}}\rangle]$. By protocol selection rule I.2b, P3($\bar{I}$, $\bar{J}$) must be obeyed. Hence, $j_1 \text{->}_{wr} i_1$ implies $j_1 < i_1$, and by transitivity $i_0 < i_1$.

In the next three subcases (d,e,f), assume $i_0 \text{->}_{wr} j_0$.

Subcase d: Suppose $j_1 \text{->}_{ww} i_1$. Then $j_1 < i_1$ and T'_{CG} is the nonredundant cycle $[\langle w^{\bar{I}}, r^{\bar{J}}\rangle, \langle r^{\bar{J}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, w^{\bar{I}}\rangle]$. By the protocol selection rule I.2b,, P3($\bar{J}$, $\bar{I}$) must be obeyed. Hence, $i_0 \text{->}_{wr} j_0$ implies $i_0 < j_0$, and by transitivity $i_0 < i_1$.

Subcase e: Suppose $j_1 \text{->}_{rw} i_1$. By the protocol selection rule II, P1($\bar{J}$, $\bar{I}$) must be obeyed. Hence, $i_0 \text{->}_{wr} j_1 < j_1 \text{->}_{rw} i_1$ implies $i_0 < i_1$.

Subcase f: Suppose $j_1 \text{->}_{wr} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{I}}, r^{\bar{J}}\rangle, \langle r^{\bar{J}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, r^{\bar{I}}\rangle, \langle r^{\bar{I}}, w^{\bar{I}}\rangle]$, and both P3($\bar{I}$, $\bar{J}$) and P3($\bar{J}$, $\bar{I}$) must be obeyed. P3($\bar{J}$, $\bar{I}$) and $i_0 \text{->}_{wr} j_0$ implies $i_0 < j_0$. P3($\bar{I}$, $\bar{J}$) and $j_1 \text{->}_{wr} i_1$ implies $j_1 < i_1$. So, by transitivity, $i_0 < i_1$.

In the final three subcases (g,h,i), assume $i_0 \to_{rw} j_0$.

Subcase g: Suppose $j_1 \to_{ww} i_1$. Then $j_1 < i_1$ and T'_{CG} is the nonredundant cycle $[\langle r^{\bar{I}}, w^{\bar{J}}\rangle, \langle w^{\bar{J}}, w^{\bar{I}}\rangle, \langle w^{\bar{I}}, r^{\bar{I}}\rangle]$. So, P3($\bar{I}$, $\bar{J}$) must be obeyed. Since $i_0 \to_{rw} j_0$, $i_0 < j_0$. Hence, by transitivity, $i_0 < i_1$.

Subcase h: Suppose $j_1 \to_{rw} i_1$. This case is essentially the same as subcase f.

Subcase i: Suppose $j_1 \to_{wr} i_1$. Then, P1($\bar{J}$, $\bar{I}$) must be obeyed. We assume $i_1 < i_0$ and show a contradiction. This follows directly, since $j_1 \to_{wr} i_1 < i_0 \to_{rw} j_0$ implies $j_1 < j_0$, contradicting $j_0 \leq j_1$.

Case III Assume that n is greater than two. We define a class, $\bar{I}$, to be a *P2-class* in T if there are transactions i and i' in $\bar{I}$ that appear in T in edges of the form $j \to_{wr} i$ and $i' \to_{rw} k$, for some transactions j and k. (Note that classof(i_0) = classof(i_{2n-1}) can be a P2-class.) We first prove two preliminary claims about T.

Claim 3: Let $[\langle j_0, j_1\rangle, \ldots, \langle j_{2m-2}, j_{2m-1}\rangle]$ be a sequence of contiguous edges in T(i.e., a subsequence of T) such that no j_k is in a P2-class of T. Then $j_0 < j_{2m-1}$.

Proof of claim 3: We prove the claim by induction on the number of edges in the sequence. As the basis, we show j_0

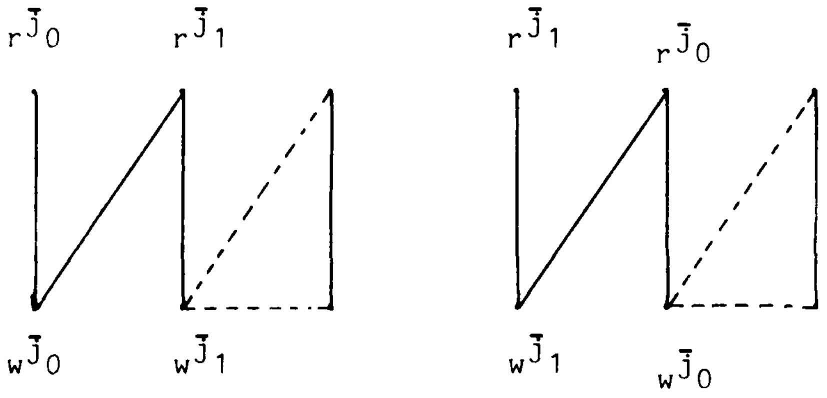
$< j_1$. Then we show that after adding an edge to the prefix $[\langle j_0, j_1\rangle, \ldots, \langle j_{2p-2}, j_{2p-1}\rangle]$, where p is smaller than m, if $j_0 < j_{2p-1}$, then $j_0 < j_{2p+1}$.

Basis step (see Figure 4.5): Either $j_0 \rightarrow_{ww} j_i$, $j_0 \rightarrow_{wr} j_1$, or $j_0 \rightarrow_{rw} j_1$. If $j_0 \rightarrow_{ww} j_1$, then $j_0 < j_1$ follows directly from the definition $\rightarrow_{ww}$. Suppose $j_0 \rightarrow_{wr} j_1$. Then the diagonal edge $\langle w^{\bar{j}_0}, r^{\bar{j}_1}\rangle$ appears in T'_{CG} and, with claim 2, T'_{CG} is a nonredundant cycle with a diagonal edge. Since j_1 is not in a P2-class, the edge following $\langle j_0, j_1\rangle$ (or edge $\langle i_0, i_1\rangle$, if $j_1 = i_{2n-1}$) must be an $\rightarrow_{ww}$ or $\rightarrow_{wr}$. So, $P3(\bar{j}_1, \bar{j}_0)$ must be obeyed. Hence, $j_0 \rightarrow_{wr} j_1$ implies $j_0 < j_1$. Suppose $j_0 \rightarrow_{rw} j_1$. The diagonal edge $\langle r^{\bar{j}_0}, w^{\bar{j}_1} \rangle$ appears in T'_{CG} and, with claim 2, T'_{CG} is a nonredundant cycle with a diagonal edge. Since $\bar{j}_0$ is not a P2-class, the edge preceding $\langle j_0, j_1\rangle$ (or edge $\langle i_{2n-2}, i_{2n-1}\rangle$ if $j_0 = i_0$) must be an $\rightarrow_{ww}$ or $\rightarrow_{rw}$. So, $P3(\bar{j}_0, \bar{j}_1)$ must be obeyed. Hence, $j_0 \rightarrow_{rw} j_1$ implies $j_0 < j_1$.

Induction step: Suppose $[\langle j_0, j_1\rangle, \ldots, \langle j_{2p-2}, j_{2p-1}\rangle]$ has $j_0 < j_{2p-1}$. If $p = m$, then the claim is proved. Else, augment the sequence by the edge $\langle j_{2p}, j_{2p+1}\rangle$. By the same argument as the basis step, $j_{2p} < j_{2p+1}$. By definition of trail, $j_{2p-1} \leq j_{2p}$. So, by transitivity, $j_0 < j_{2p+1}$, thereby proving the claim.

Step of Claim 3 of Lemma Trail

Figure 4.5



$j_0 \rightarrow_{wr} j_1$ $j_0 \rightarrow_{rw} j_1$

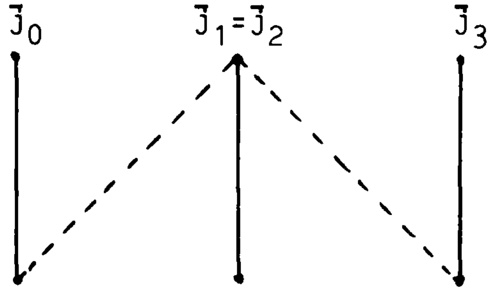
Claim 4: Let $[\langle j_0, j_1\rangle, \langle j_2, j_3\rangle]$ be a sequence of contiguous edges in T such that $j_0 \rightarrow_{wr} j_1$ and $j_2 \rightarrow_{rw} j_3$. Then $j_0 < j_3$.

Proof of claim 4(see Figure 4.6): Since T'_{CG} contains the diagonal edge $\langle r^{\bar{j}}1, w^{\bar{j}}0\rangle$ and $\langle r^{\bar{j}}1, w^{\bar{j}}3,$, the protocol selection rules imply that $P2(\bar{j}_1,\{\bar{j}_0, \bar{j}_3\})$ must be obeyed. So $j_0 < j_3$ follows by definition of P2, proving the claim.

We can prove case III for two subcases: $\bar{i}_0$ is a P2-class and $\bar{i}_0$ is not a P2-class.

Suppose $\bar{i}_0$ is a P2-class. Consider T with its first and last edges removed (call it T"), T" = $[\langle i_2, i_3\rangle,\ldots,\langle i_{2n-4}, i_{2n-3}\rangle]$. (Since n is greater than two, there must be at least one edge in between.) T" consists of sequences with no P2-class transactions (as per claim

Claim 4 of Lemma Trail Figure 4.6



3) separated by sequences of P2-class transaction pairs. By claims 3 and 4, the left and right endpoints of each of these sequences satisfy the relation left-endpoint < right-endpoint. So, by repeated application of transitivity, we have $i_2 < i_{2n-3}$. Since $\bar{I}_0$ is a P2-class, P2($\bar{I}_0$, {$\bar{I}_2$, $\bar{I}_{2n-3}$}) must be obeyed. By examining T, we have $i_0 ->_{rw} i_1 \leq i_2 < i_{2n-3} \leq i_{2n-2} ->_{wr} i_{2n-1}$. Suppose $i_{2n-1} < i_0$. Then $i_{2n-2} ->_{wr} i_{2n-1} < i_0 ->_{rw} i_1$, $\bar{I}_0 = \bar{I}_{2n-1}$, $\bar{I}_2 = \bar{I}_1$, and $\bar{I}_{2n-2} = \bar{I}_{2n-3}$ implies (by P2) that $i_{2n-2} < i_1$. But $i_1 \leq i_2 < i_{2n-3} \leq i_{2n-1}$ implies $i_1 < i_{2n-1}$, a contradiction. So $i_0 < i_{2n-1}$.

Suppose $\bar{I}_0$ is not a P2-class. Then T has exactly the same form as T" above. So, by repeated application of claims 3 and 4 and transitivity, we obtain $i_0 < i_{2n-1}$ as desired. This proves case III and the lemma. Q.E.D.

Lemma TRAIL-P4

Let H be a well-behaved execution history and let -> and $SG_0(H)$ be defined on H. Let $t = [\langle i_0, i_1\rangle, \ldots, \langle i_{2p}, i_{p+1}\rangle \ldots, \langle i_{2n-2}, i_{2n-1}\rangle]$ be a trail in $SG_0(H)$ such that i_{2p} ran P4 with respect to all classes that have transactions on the trail. Then $i_0 < i_{2n-1}$.

Proof

We show that for each q where $0 \leq q \leq p$ we have $i_{2q} < i_{2p}$, and for each q where $p < q \leq n$ we have $i_{2p} < i_{2q-1}$. So, $i_0 < i_{2p} < i_{2n-1}$ as desired.

Let $[\langle i_{2q+2}, i_{2q+3}\rangle, \ldots, \langle i_{2p}, i_{2p+1}\rangle]$ be a subtrail of T where $0 \leq q < p$ and $i_{2q+2} < i_{2p}$. Consider $\langle i_{2q}, i_{2q+1}\rangle$. Since $i_{2q} \rightarrow i_{2q+1} \leq i_{2q+2} < i_{2p}$, by P4 $i_{2q} < i_{2p}$. So, $[\langle i_{2q}, i_{2q+1}\rangle, \ldots, \langle i_{2p}, i_{2p+1}\rangle]$ is a subtrail of T with $i_{2q} < i_{2p}$. By a simple induction argument, $i_{2q} < i_{2p}$ for each q, $0 \leq q < p$.

Let $[\langle i_{2p}, i_{2p+1}\rangle, \ldots, \langle i_{2q-4}, i_{2q-3}\rangle]$ be a subtrail of T where $p < q \leq n$ and $i_{2p} < i_{2q-3}$. Consider $\langle i_{2q-2}, i_{2q-1}\rangle$. Since $i_{2p} < i_{2q-3} \leq i_{2q-2} \rightarrow i_{2q-1}$, by P4 $i_{2p} < i_{2q-1}$. So, $[\langle i_{2p}, i_{2p+1}\rangle, \ldots, \langle i_{2q-2}, i_{2q-1}\rangle]$ is a subtrail with $i_{2p} < i_{2q-1}$. Again, a simple induction argument shows $i_{2p} < i_{2q-1}$ for each q, $p < q \leq n$, and the lemma is proved. Q.E.D.

Lemma PATH Let H be a well-behaved execution history and let -> and $SG_0(H)$ be defined on H. Let $P = [<i_0, i_1>,\ldots,<i_{2n-2}, i_{2n-1}>]$ be a path in $SG_0(H)$, where $\text{classof}(i_0) = \text{classof}(i_{2n-1})$. Then $i_0 < i_{2n-1}$.

Proof

Our proof is by induction on n. As the basis step, assume that n = 1. Then by definition of ->, P is a trail. If $\text{classof}(i_0)$ is not a P-class, then $i_0 < i_{2n-1}$ follows from lemma TRAIL. If $\text{classof}(i^0)$ does run P4, then $i_0 < i_{2n-1}$ follows directly from the definition of P4.

Assume the lemma is true for all n less than k. We now show it to be true for n = k. If P is a trail, the $i_0 < i_{2n-1}$ by lemma TRAIL of Lemma TRAIL-P4 and we are done. So, assume P is not a trail. Then there is some class $\bar{J}$ that contains transactions j' and j" in P that are connected by a nonempty proper subpath of P. There are two cases: for some such $\bar{J}$, $\bar{J}=\bar{I}_0$ and for no such $\bar{J}$ does $\bar{J}=\bar{I}_0$.

If $\bar{J}=\bar{I}_0$, then there is a q between 1 and n-1 with $\bar{I}_0=\bar{I}_{2q}$. So, P can be partitioned into two paths $[<i_0, i_1>,\ldots,<i_{2q-2}, i_{2q-1}>]$, and $[<i_{2q}, i_{2q+1}>,\ldots,<i_{2n-2}, i_{2n-1}>]$. By the inductive assumption, $i_0 < i_{2q-1}$ and $i_{2q} < i_{2n-1}$, since the subpaths are of length less than k. So, $i_0 < i_{2n-1}$ by transitivity, and we are done.

If $\bar{J} \neq \bar{I}_0$, then P is of the form $[\langle i_0, i_1\rangle, \ldots, \langle j', \rangle, \ldots, \langle , j"\rangle, \ldots, \langle i_{2n-2}, i_{2n-1}\rangle]$. Excise the path from j' to j" from P. By the induction assumption, j' < j". Excise all such paths that connect two transactions in the same class. When no such transaction pairs remain (except those that are adjacent), the resulting sequence of edges is a trail. (Adjacent edges are incident with two transactions from the same class that are either identical or are related by increasing <. Also, no class appears in more than two heterogeneous edges.) Now, by lemma TRAIL, $i_0 < i_{2n-1}$, as desired. Q.E.D.

Lemma ACYC follows as a corollary to lemma PATH.

<u>Lemma ACYC</u> If execution history H is well-behaved, then $SG_0(H)$ is acyclic.

<u>Proof</u>

Suppose there is a path in $SG_0(H)$ from transaction i back to itself. By lemma PATH, i < i. But < is a total order, a contradiction. Q.E.D.

We may now state and prove the main theorem of this section:

<u>Theorem SR</u>: If all execution histories produced by SDD-1 are well-behaved, then they are all serializable.

Proof: Follows directly from corollary TOSG and lemma ACYC. Q.E.D.

5. Serializability of Logical Transactions

The formalism which has been presented so far deals with transactions which have read-sets and write-sets of physical data items. In practice, however, the user of SDD-1 expresses his transactions in terms of logical data items. A logical data item may correspond to a number of physical copies stored in the database, presumably all at different sites. SDD-1 maps user transactions expressed in terms of logical data items into transactions referring to physical data items according to the following rule. When a logical data item is read the system chooses one or more of the physical copies to read. However, when a logical data item is written, the system updates every physical copy of the logical data item.

We would like to prove that SDD-1 generates a serializable history of logical transactions against a logical database. That is, the transactions appear to be serializable against a hypothetical database in which there is only one copy of each logical data item. Furthermore, we wish to show that the write message rule is not needed in the serialization, i.e. all write messages always apply all of their updates in the serialization. That is, updates specified in the user's transaction always update the database, and cannot be accidently ignored due to the write message rule.

We could extend our formalism to include the notions of logical transaction, logical data item, a logical to physical data item mapping, and a correctness definition based on logical transactions rather than physical transactions. However, instead of actually developing this additional mechanism we will simply present an informal plausibility argument for the serializability of logical transactions.

The argument is a simple one and goes as follows. Consider the serialized execution history corresponding to an actual interleaved history. Between completely executed transactions in this serial history, all physical copies of each logical data item have the same value. This follows because any transaction which updates one copy must update all the others as well. Thus, since all physical copies have the same value, the behavior of the system is the same as if there were only one physical copy corresponding to each logical data item. And further, since the actual interleaved hisory is defined to be equivalent to the serialized history, it too behaves as if each logical data item had only a single physical copy.

Finally, we note that whenever a write-write intersection occurs between transactions, the system serializes those transactions in <-order. Therefor the serial history behaves as if all updates applied unconditionally to the database, without reference to the write message rule. And thus, by equivalence, all transaction updates actually affect the database.

Acknowledgments

Special thanks are due to E. Cardoza and M. Casanova for correcting a number of errors in previous drafts.

References

[BERNSTEIN et al.]
Bernstein, P.A.; Rothnie, J.B.; Goodman, N.; and Papadimitriou, C.A. "Concurrency Control in SDD-1: A System for Distributed Databases; Part 1: Description", submitted to 1979 Transactions on Database Systems.

[ESWARAN et al.]
Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[GRAY et al.]
Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", Report from IBM Research Laboratory, San Jose California, 1975.

[HEWITT]
Hewitt, C.E. "Protection and Synchronization in Actor Systems", Artificial Intelligence Laboratory Working Paper No. 83, Massachusetts Institute of Technology, November 1974.

[MANNA]
Manna, Z. Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

[PAPADIMITRIOU et al.]
Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

RELIABILITY MECHANISMS FOR SDD-1:

A SYSTEM FOR DISTRIBUTED DATABASES

Michael Hammer

David Shipman

Computer Corporation of America

and

Massachusetts Institute of Technology

Cambridge, MA 02139

July 31, 1979

ABSTRACT

This paper presents the reliability mechanisms of SDD-1, a prototype distributed database system being developed by the Computer Corporation of America. Reliability algorithms in SDD-1 center around the concept of the Reliable Network (RelNet). The RelNet is a communications medium incorporating facilities for site status monitoring, event timestamping, multiply buffered message delivery, and the atomic control of distributed transactions. This paper is one of a series of companion papers on SDD-1 [Rothnie et al, Bernstein et al, Bernstein and Shipman, Wong].

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract Number N00039-77-C-0074, ARPA Order Number 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the United States Government.

1. INTRODUCTION

1.1. The RelNet

This paper describes the Reliable Network, a subsystem of the SDD-1 distributed database management system, whose function it is to provide the level of reliability and robustness demanded of a system that is charged with responsibility for an organization's data. One of the prime motivations for building a distributed system is to achieve reliability enhanced over that which would be provided by a single-site system; the redundancy of data and processors provided by a distributed system potentially enable it to continue in operation despite the failure of individual sites. (In a single-site system, of course, site failure causes the entire system to cease operation.) However, although a multi-site system presents opportunities for enhanced reliability, it also presents challenges in the same area, because the likelihood that some part of the total system will fail becomes much higher. The goal is to design and implement distributed systems that exhibit global robustness and toleration of local site failures, that can continue to operate as a whole despite the asynchronous failures and potential recoveries of individual elements of the system.

The Reliable Network (known as the RelNet) has been designed to provide a set of capabilities to support such reliable distributed system operation. Its original conception was in the context of the SDD-1 system, and some of its features were motivated by the particular requirements of that system. However, we believe that the functional capabilities of the RelNet have wide applicability to distributed systems of many kinds; it may, in fact, represent a forerunner of a general reliable distributed operating system. This paper describes the functionality, architecture, and implementation techniques of the RelNet. The particular ways in which the RelNet is used to support reliable operation of SDD-1 are described in [Bernstein et al].

The RelNet consists of a set of facilities intended to ensure reliable communication and coordination among related processes operating at sites connected by means of a communications network. In a distributed system, a function will in general be realized by means of a number of processes, executing in parallel at distinct sites of a network. As execution of these processes proceeds, they will find occasion to communicate and synchronize with each other. The designer of a distributed system will have to recognize the reality that individual sites and processes in this system may fail at any point in time; consequently, each site must be prepared to recognize and react to the failure(s) of its "cohorts" [Gray], the other sites with which it cooperates and interacts. One approach would be to embed this responsibiiity in the application logic and code of each cohort. However, following general principles of good software design, it would be preferable to provide the application programmer with a (fictional) view of the environment, which exhibits a degree of reliability that simplifies his system design and implementation. This view is the RelNet. The RelNet provides each process running in the system with a set of facilities for reliable communication and interaction with other processes; these facilities can be utilized by invoking a set of procedure calls. Thus, the RelNet is used instead of whatever communication facilities are provided by the actual communication network connecting the sites in the distributed system. Many implementations of the RelNet facilities would be possible; we have chosen to realize it "on top" of the real network, by means of a software front-end on each machine in the distributed system.

This front-end represents an interface to the communication network, intervening between application programs and the operating/communication system.

1.2. RelNet Facilities

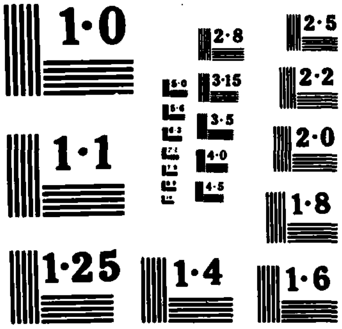
The basic function of any network is to allow for inter-site communication. The RelNet can be effectively thought of as a virtual network that provides the following additional capabilities.

1. There exists within the network a single Global Clock that can be accessed from any site. The function of such a clock is to impose a uniform and consistent ordering on events occurring at different sites in a distributed system. The current value of the clock can be inspected by a user process.

2. Every site in the network is at any time in one of two states, UP or DOWN. The UP state is characterized by correct operation and by timely response to messages sent by other sites; a site in the DOWN state is not operating at all. Transitions between these states (called "crashes" and "recoveries") occur instantaneously with respect to the global clock. A user process is provided with the ability to ascertain the current status of any site in the network, and to request that it be informed when that site changes its state.

3. The RelNet provides a reliable communication service that makes two guarantees. First, that messages sent from one site to another are received in the same order that they are sent. Second, that, on user request, a message can be marked for guaranteed delivery. That is, a message can be sent to a site that is

AD-A155 724   A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND   3/4
AND CONTROL APPLICATIONS PART 2(U) COMPUTER CORP OF
AMERICA CAMBRIDGE MA 30 JAN 80 CCA-80-04-PT-2
UNCLASSIFIED   N00039-77-C-0074   F/G 9/2   NL


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

DOWN, with the RelNet guaranteeing that the message will be received by that site upon its recovery. Receipt takes place even if the sending site is DOWN at the time the destination site recovers.

4. A facility for distributed transaction control is provided. This provides for a process running at one site to coordinate the activities of a number of "cohort" processes that are running at different sites and seeking to realize a global activity. The principal feature of this facility is its global abort/commit capability, which enables the controlling process to instantaneously cancel the transaction at any point or to signal its successful completion and cause the results to uniformly take effect at all involved sites.

1.3. Layered Architecture

The RelNet is itself organized and implemented as a series of software layers, each of which provides a subset of the facilities as a whole; furthermore, the lower layer capabilities are utilized in implementing those of the higher levels. The lowest level of the RelNet is known as the Global Time Layer, and provides the global clock and site status features described above. The next layer is Guaranteed Delivery, which enables a user to send messages to down sites, with the assurance they will be received when the site recovers. The topmost layer of the RelNet is the Transaction Control layer, which provides the ability to manage and coordinate a distributed transaction and deal with it atomically. We believe that this layered architecture contributes to the comprehensibility and implementability of the system as a whole.

1.4. Catastrophes

The RelNet is a system constructed out of discrete components: sites and communications lines, each of which is subject to failure. It is our goal to achieve the desired level of reliable functionality by techniques that will allow for failures of some number of these components. In other words, the RelNet is designed to be resilient to the failure of some of its parts, and to function correctly so long as enough of the components behave correctly. However, no multi-component system can survive the failure of too large a number of its constituents, and the same is true of the RelNet. When "too many" failures occur, the result is termed a catastrophe. Under catastrophe situations, the RelNet is not guaranteed to operate correctly. In some cases, it will simply fail to provide one of its functions, while in others it may operate in unanticipated and unpredictable ways. Although some catastrophe situations can be automatically detected by the RelNet, others can only be observed from outside the system. In either case, manual intervention by a system administrator or other responsible human authority is necessary in order to rectify the situation. This will often entail shutting down part of the system and reinitializing it. The various catastrophes that can befall the RelNet facilities are described together with the individual mechanisms that realize them.

It is a principle of the RelNet design that, although catastrophes can not be entirely avoided, they can be made arbitrarily unlikely by the increased replication of reliability mechanisms. In other words, the reliability of the RelNet is parameterizable in terms of such factors as the number of sites preforming various backup functions. The price to be paid for such increased reliability is of course commensurate increased overhead. The tradeoff between the two is one that must be made by the system administrator, depending on such factors as the requirements of his application and the individual reliabilities of his system components. Having made his decision, the system administrator implements it by selecting values for a number of parameters that are identified in the paper.

1.5. Assumptions

The components out of which the RelNet is constructed are computer sites and communications lines connecting them. We assume that the communications lines are already organized into a basic computer communication network, with a conventional set of capabilities. Consequently, failures of individual communications lines are not explicitly considered or addressed by the RelNet; they are the province of the underlying network. That is, the RelNet assumes that the underlying network will detect the failure of a communication line between two sites and employ others to send messages between them. Furthermore, the RelNet assumes that the network at all times remains fully connected; that is, that there is at all times some path of communications lines between any two sites in the system. Should this assumption be violated by the failure of key communications lines that result in certain sites being disconnected from others, the result is a RelNet catastrophe, known as a partition catastrophe. Techniques for detecting and coping with such a situation are dealt with in the Appendix. Other communications failures are not addressed in this paper. Further assumptions about the capabilities of the communications network are described in Section 2.

We also make assumptions about the way in which the computer sites of the system may fail. First, we assume that every site is equipped with a "stable storage" mechanism. This is a device that enables any site to ensure that critical information that it has received can be stored in a way that will enable it to survive a failure of the site. Techniques for implementing stable storage are discussed in [Lampson and Sturgis, Verhofstad, Lorie]. We do allow for failures to occur asynchronously, and at any point in the system's operation. However, we do by and large restrict our attention to "clean" failures, in which the site completely ceases operation. We further assume that a site that is recovering from failure is aware of that fact and can be made to institute suitable recovery procedures.

1.6. Relation to Previous Work

A number of other researchers have considered the problem of reliability in a distributed system [Lampson and Sturgis, Svobodova, Gray, Alsberg and Day, Reed, Montgomery, Schapiro and Millstein]; however, the RelNet differs significantly from their work. In the first place, the RelNet is, to the best of our knowledge, the only attempt thus far to develop a complete and integrated facility for reliable distributed database system operation, one which addresses the full range of issues that arise in that context. Individual problems have been studied in isolation, but the interactions among them had not been previously explored. Furthermore, previously proposed solutions to specific reliability problems are not directly applicable to the SDD-1 environment; this is primarily a consequence of our insistence that no transaction be forced to delay its execution until another site has recovered from failure (except when the failed site possesses the only copy of some data that the transaction requires). In some cases, we have been able to build on earlier work, while in others entirely new mechanisms were required. Finally, a number of the facilities and features of the RelNet have been motivated by the particular needs and requirements of SDD-1; this has led to a specific structure for our reliability techniques, which is based on a global clock mechanism. This orientation lends a unique character to the RelNet architecture.

1.7. Structure of the Paper

This paper seeks to set forth the basic facilities of the RelNet and a particular set of implementation mechanisms that can be used to realize them. Our presentation follows the layered architecture of the RelNet itself. For each layer, we identify the functionality that it provides and then describe its implementation. In general, the implementation of each layer will be expressed in terms of the facilities provided by the lower layers of the system. There are two aspects to the implementation of each RelNet mechanism; the first concerns how sites are to behave under normal operation, and the second how they react to the failure and recovery of sites (including themselves). Consequently, an important part of the RelNet implementation is concerned with how a recovering site manages to bring itself back into normal operation. In the RelNet, site recovery is also a layered operation, corresponding to the layers of RelNet implementation. Thus, a recovering site will first execute the lowest level of recovery mechanism, then the next, and so on, until it has completed the entire process; at that point, it is considered to be fully recovered.

Distributed system reliability is an extremely intricate problem. Herein, we focus on the fundamental concepts of the mechanisms employed in the RelNet, and indicate by footnote where they need extension in order to handle special cases.

2. THE MESSAGE TRANSMISSION LAYER

The lowest level of the RelNet consists of the basic message transmission facilities of the underlying computer network on top of which the RelNet is constructed. For our purposes, we shall assume that this layer provides facilities for sending and receiving messages between sites. However, no guarantees are made by this layer that a message that is sent by one site will actually be received at the intended destination. In particular, a message is lost if the receiver fails before the message arrives. The only guarantee that the Message Transmission Layer does provide is the following. If receiver A receives two messages from sender B, then they are received in the order in which they were sent; furthermore, if A did not fail between the receipt of these two messages, then it also received any other messages sent it by B between these two. The failure to meet this guarantee is a RelNet catastrophe, which can have a variety of impacts on the higher levels of the RelNet and on SDD-1 user processes.

In general, the only way for a sender to be sure that a message has been received by the destination is to require an acknowledging response from the destination site. Such acknowledgement protocols, however, are not provided by the Message Transmission Layer, but are the responsibility of programs that use it. In particular, higher levels of the RelNet expect to receive various responses to or acknowledgments of the messages that they send. These responses are typically returned by the same level of the system that is responsible for issuing the original message. Frequently, the sender will employ a time-out mechanism to limit the amount of time it will wait for such a response; if no such response is received within this period, the sender will assume that the intended recipient is down and will take appropriate action. We shall see several instances of this in subsequent sections of this paper.

3. THE GLOBAL TIME LAYER

3.1. Introduction

Since it is a distributed system, SDD-1 must possess some mechanism to allow it to coordinate and synchronize actions being performed at different sites in a network. It employs a global clock mechanism for this purpose. A clock is simply a uniformly increasing counter whose values can be associated with events; this technique is known as event timestamping. Timestamps must be consistent with the order of occurrence of events, so that a later event has a greater timestamp. Among the events that need to be consistently ordered by timestamps are the sending and receiving of messages between sites and the failure and recovery of sites. The principal function of the Global Time Layer of the RelNet is to provide SDD-1 with such a consistent and accurate global clock mechanism. In order to do so, it must also encompass the functions of message transmittal and reception and the monitoring of site status.

Specifically, the Global Time Layer presents to higher levels of the RelNet and to SDD-1 a virtual network with the following characteristics: there exists with the network a single global clock, by means of which events at any site in the system can be timestamped and thereby ordered; that every site in the network is at any time in one of two states, UP or DOWN; and that transitions between these states, called crashes and recoveries, occur instantaneously with respect to the global clock. The interface to the Global Time Layer provides higher levels of the RelNet and user processes the following abilities: to send a message to another site, which will be timestamped with the value of the global clock at the time that it is sent; to receive a message; to determine the value of the global clock; to determine the current status of any site in the network; and to request that a "Watch" be placed on any site, so that the requesting process is notified when that site changes its status.

The central concept in the Global Time Layer is that of a global clock; this is a mechanism used to achieve an ordering of events occurring in a distributed system. [Lamport] observes that events in a distributed system should be considered to be partially ordered rather than totally ordered. That is, a relative order need be established between events in different processes only if there is some communication between the processes that could serve to pass information about event occurrence from one to the other and thus enable knowledge of one event to influence the other. A global clock can be used to order events, by associating with each event the value of the clock at the time of its occurrence. The clock value associated with an event is known as its timestamp. The absolute value of a timestamp is of no interest; only the relative values of those of ordered events need concern us. (In other words, the principal requirement of a global clock is that it support and model our notion of causality.) The following rules determine when two events need have a defined order:

1. Events within a single process are totally ordered by their execution sequence.

2. If process B learns of event1 in process A before performing event2, then event1 must precede event2.

In this section, we shall see how the Global Time Layer is implemented by means of local facilities (especially local clocks and status tables). The design task with which we are faced is to coordinate local clocks and local status table in such a way as to present an interface that will simplify the design of procedures operating outside the Global Time Layer.

The implementation of the Global Time Layer is itself a layered one; however, the lower layers do not necessarily provide coherent and useful packages of capabilities, but are designed so as to realize a structured implementation of the Global Time Layer. Each layer in the implementation performs its functions by calling upon the facilities provided by lower levels. In some cases, similar facilities are provided at several levels; for example, each layer has Send and Receive primitives. However, the clean separation of the layers should contribute to the system's understandability and implementability.

3.2. The Local Clock Layer

The first level of the Global Time Layer is the Local Clock Layer. This implements a local clock facility that is used by higher levels of the system to simulate a uniform and consistent network-wide global clock. A local clock is simply a monotonically increasing counter that is logically independent of any real time measurement. The local clock can be read to provide timestamps for events occurring at the site. In particular, a timestamp will be assigned to every message sent from one site to another. These timestamps are constructed by appending the current local clock value (as high-order bits) to the unique site identifier (as low-order bits). After a timestamp has been requested, the clock value will be incremented, so that the next timestamp assigned will differ from the last one.

In brief, the interface to the Local Clock Layer consists of the following functions:

Readclock(), which returns the current value of the local clock.

Bumpclock(n), which increments the value of the clock to be greater than the value n (this has no effect if the clock value is already greater than n).

Sendmsg(m,d), which assigns a timestamp to the message m and dispatches it to its destination d.

Receive(), which receives a (timestamped) message.

In order to support the global ordering of events demanded by a global clock, the Local Clock Layer will examine the timestamp of each message that it receives. If its value is greater than that of the current local clock, then the Local Clock Layer will bump the local clock beyond the timestamp value. In this way, the (local) time at which a recipient receives a message is greater than the (local) time at which the sender sent it. (This capability is necessary but not sufficient for implementing a full global clock; the remaining issues are dealt with in a subsequent section.)

During site recovery from failure, the Local Clock Layer simply sets the value of the local clock to be 0. Subsequent receipt of timestamped messages from other sites and attendant clock manipulations will restore the local clock to an appropriate value.

In addition to the logical clock facilities just described, the RelNet depends on the existence of a local real-time clock at each site. This clock is principally used to implement a time-out feature, by means of which a site that does not respond to a message within a given period is assumed to be no longer operating. In general, there need be no relationship between a site's real-time and logical clocks; the former is used to measure actual time, while the latter is for timestamping events. However, in some cases (described in a subsequent section), it is desirable that over a period of elapsed real-time, an equal amount of logical time should also have elapsed. To this end, the two clocks are kept in rough synchrony by the following technique. The two clocks are commensurable, in that they both employ the same units. The real-time clock is coarse-grained, the logical clock fine-grained. E.g., each time a new timestamp is required, the value of the logical clock will be incremented by 1, while each half-second (say), the real-time clock will be incremented by 10,000 (say). In addition, each time the real-time clock is updated, the logical clock is bumped past the new value of the real time clock. That is, if the new value of the real-time clock is greater than that of the logical clock, the latter is pushed beyond that value. The value by which the real-time clock is incremented is chosen to be large enough so that as a rule, the logical clock will not advance that much on its own within the interval between real-time clock advances. (Occasional lapses from this assumption are not critical.) The net effect of this mechanism is to bound the discrepancy between the logical and the real-time clocks. In this way, elapsed real-time can be roughly measured on the logical clock when necessary.[1]

This rough synchrony between logical and real clocks has two additional desirable consequences. First, it keeps the logical clocks at different sites approximately synchronized (assuming that their real-time clocks are). This is desirable for reasons of efficiency in the SDD-1 concurrency control mechanisms [Bernstein et al]. In addition, in the event of a catastrophe that requires logical clocks to be manually reset, the system administrator can employ the real-time clock in this process.

3.3. The Local Status Layer

The Local Status Layer provides a set of primitives for manipulating and utilizing a site's Local Status Table, which is used to represent the site's view of the condition of all other sites in the network. These primitives are used to support the simulation of the global clock and in informing user processes of the current status of sites with which

[1] It should be noted that the basic mechanism presented above is vulnerable if some site has a "runaway" logical or real-time clock. Additional mechanism can be used to resolve this problem.

they seek to communicate.

For each site in the network (including itself) a site's Local Status Layer maintains an entry in the Local Status Table that specifies the condition of that site. (The entry for the site itself is handled in a special way, as discussed below.) Entries are made into the table upon instruction from higher layers of the Global Time Layer implementation. The only status values that may be entered for a site are UP and DOWN.

The interface to the Local Status Layer provides the following capabilities:

1. Sending and receiving messages.

2. Setting the status value of any site (via MarkUP and MarkDOWN primitives).

3. Requesting that a Watch be placed on a given site (or removed from it).

The Watch facility enables higher levels of the RelNet and user processes to state that they wish to be informed when a given site achieves a specified status. (Thus, there are two types of Watch; one which waits for a site to become UP and one for it to go DOWN.) The Local Status Layer will determine when that situation obtains and interrupt the requesting process to inform it that the condition has been met. (If the site is already in the designated state when the Watch is issued, the call to set the Watch will immediately return.)

The Local Status Table has an entry for each site in the network, which states whether that site has last been observed to be UP or DOWN. Furthermore, flags may be set to indicate if a Watch has been set on the site and by which user process.

Detection of site status change (either a crash from UP to DOWN or a recovery from DOWN to UP) is not performed by the Local Status Layer. The Local Status Layer is only responsible for managing the status table; all changes to a sites status are initiated from higher levels of the Global Time Layer. When such a change is detected, the Local Status Layer is instructed (by means of the MarkDown and MarkUp primitives) to change the appropriate entry in the Local Status Table. It is at that this point that any processes that have requested a watch against the site will be interrupted.

When the Local Status Layer is given a message to send to a site, it first inspects the Local Status Table entry for that site. If the site is listed as DOWN, it discards the message and informs the issuing process that the message could not be delivered because the destination site is down. (The sending process could then decide to take other steps, or it could issue a recovery watch on the site in question and resend the message when that event occurs.) If the site is listed as UP, the Local Status Layer sends the message to the destination, using the send primitives of the Local Clock Layer.

All messages destined for user processes or higher levels of the RelNet also pass through the Local Status Layer. Upon receiving a message, the Local Status Layer will first check the status of the sending site. (Let us call the sending site A and the receiving site B.) If the site A is marked as UP at B, then the message is simply passed through. If the site is marked DOWN, then the nature of the message is examined; one message type is handled differently from all others. An I'M-UP message is sent out by a site upon its recovery; such messages are sent and processed by a higher level of the RelNet. When B's Local Status Layer receives an I'M-UP message from A (a site that it has marked as DOWN), it passes through the message to be processed by a higher level of the system. If B's Local Status Layer receives any other kind of message from A, then B had incorrectly assumed that A was down (when in fact it had probably only been slow to respond to some earlier message). However, B may already have taken some action based on the assumption that A was down; consequently, it must force A to indeed be down. This is accomplished by sending A a YOU'RE-DOWN message. This message, when processed by A's Local Status Layer, will cause it to behave as though it had indeed failed, thereby validating the assumption made by B. (Subsequent recovery of A will then be instituted.)

In addition to issuing YOU'RE-DOWN messages, the Local Status Layer also has responsibility for processing incoming YOU'RE-DOWN messages. Upon receiving such a message, it should cause the local site to crash (and then recover). (This might be accomplished simply by transferring control to the RelNet recovery module.) This should be done even if the sender of the YOU'RE-DOWN message is a site that itself is marked as DOWN. Such an occurrence indicates that two sites had made similar and mistaken assumptions about each other. In this case, the receiving site both issues a YOU'RE-DOWN message to the sender, and acts on the YOU'RE-DOWN that it received.

On recovery, the Local Status Layer simply sets the status of all sites in the network to be UP, except for its own status; that is set to be DOWN. Subsequent analyses of the responses to the I'M-UP messages (which are received and processed by a higher level of the system during recovery) will cause these values to be more accurately set.

3.4. The Global Clock Layer

The purpose of the Global Clock Layer is to present to higher levels of the RelNet and to user processes the view that the network as a whole possesses a single, uniform clock, which is used by all sites to timestamp messages and thereby to order events. That is, above this level it will appear that timestamps are assigned by a single global clock residing within the RelNet. The essential property of a global clock is that it be consistent with inter-site event ordering. This means that if an event occurs at site A at time t1, then an event that occurs at site B after B learns of the A event must occur at time t2, where t2 > t1. in reality of course, there is no "global clock"; rather, each site operates according to the timestamps issued by its own local clock. Therefore, the goal of the Global Clock Layer is to simulate a global clock by means of a collection of independently running local clocks.

The Global Clock Layer's interface provides the ability to send and receive messages; to examine and increment the value of the global clock; and to assure (by explicitly crashing it) that a site assumed to be DOWN is indeed in that state. In addition, the primitives of the Local Status Layer for manipulating the Local Status Table are passed through to higher levels of the RelNet.

The local clock mechanism described above almost realizes a global clock. That is, by timestamping every message and incrementing the local clock upon receipt of a message to be greater than the timestamp on the message, the collection of local clocks almost provides an ordering of events occurring at different sites that models their actual times of occurrence and influence on one another. However, there is a way in which the local clock facility does not truly achieve a network clock; this is caused by implicit communication through the observation of a site's failure. In our discussion of local clocks, we assumed that all communication between sites occurred through the medium of timestamped messages, and that consequently incrementing a local clock beyond the timestamp of any incoming messages sufficed to appropriately order events in the system. However, the detection of one site's failure by another also represents a form of inter-site communication, and it too must conform to the requirements of a global clock; that is, the (B-local) time at which B discovers that A has crashed must be greater than the (A-local) time at which A did crash. In other works, the detection of a site's crash is equivalent to the receipt of a (hypothetical) I-HAVE-CRASHED message, which is however untimestamped. When B detects A's failure, B should increment its local clock to be greater than the value of A's local clock at the time A failed. Unfortunately, it requires additional mechanism to enable B to determine the (A-local) time at which A failed, since A's clock is unavailable for inspection after A has crashed.

The Global Clock Layer uses the mechanisms of guardians and Timesignal messages to construct a true global clock out of a collection of local clocks. The basic concept of this mechanism is that for each site, a set of "guardians" is maintained, each of whose clocks is guaranteed to be at most a constant value less than the value of the guarded site's clock; this is achieved by means of special messages (Timesignals) sent among these sites. Then when the failure of site A is detected by site B, B will be informed by one of A's guardians of an upper bound on A's clock at the time it failed; B can then bump its own clock to be greater than this value. This will achieve the event ordering demanded by a global clock.

First we shall describe how guardians are implemented. Each site W has associated with it a fixed set of guardian sites, G1,G2,...,Gk. W is called the ward of its guardians. The system is designed to guarantee the following constraint:

> Guardian Constraint: If a site W and one of its guardians G are both UP, then Clock(G)+Tdelta > Clock(W), where Tdelta is a fixed system parameter and Clock(x) is the latest timestamp to have been issued by the local clock at site x.

This guarantee is implemented by having W maintain counters that tell it a lower bound on the clock values of each of its guardians. Before issuing a timestamp, W must check that this new timestamp can not possibly violate the guardian constraint. If it might cause its violation, then W can not issue the timestamp; instead, it will have to wait until new messages are received from the guardians that inform it that their clocks have advanced far enough so that the new timestamp will be consistent with their clocks under the terms of the guardian constraint. To avoid ever getting into such situations, in which it will have to wait before issuing a timestamp that it wishes to use, W will periodically issue "Timesignal" messages, which have the effect of keeping the clocks of W's guardians in relatively close proximity to W's own.

Specifically, for each of its guardians G, each site W maintains LBClock(G,W), which simply contains the value of the timestamp on the latest message from G received by W. (W can be sure that if G is up, then G's local clock is greater than LBClock(G,W), and if it is down, then G's clock at the time of failure was greater than LBClock(G,W).) Our goal is to ensure that W's clock does not get more than Tdelta ahead of LBClock(G,W), for each G that is up. This is accomplished by having the Global Clock Layer monitor the values of LBClock(G,W). Whenever it observes that

Clock(W) > LBClock(G,W) + Tdelta - RTMD,

where RTMD is the typical network round trip message delay, then W should send a Timesignal message to G. The Timesignal message is a timestamped but otherwise null message that requires a response. (A received Timesignal message is processed by the Global Clock Layer of the recipient site, which returns the expected response, itself a timestamped message. If no response to a Timesignal is received within a specified Timeout period, then the issuing site assumes that the guardian site has failed and invokes Crashsite against it.[1] When the condition just cited holds, it indicates that G's clock may be approaching being more than Tdelta behind W's; sending the Timesignal will

[1] Additional mechanism is required to address a situation in which a ward mistakenly believes that a guardian site has failed. This mechanism centers around special handling of YOU'RE-DOWN messages.

cause G's local clock to be incremented past the value of W's clock at the time that the signal is sent, and bring them closer into proximity. The particular value RTMD is chosen so that by the time W's own clock has advanced by RTMD, the response to the Timesignal can be expected to have been received, enabling W to increment LBClock(G,W).

(Note that there is an interaction between the logical and the real-time clocks here. The goal of the Timesignal messages is to keep the logical clocks of the guardian and the ward in approximate synchrony, yet the condition governing their issuance is expressed in terms of the average round trip message delay, a real-time quantity. In general, the elapsing of a real-time period (such as RTMD) would not necessarily coincide with an equivalent change in a site's logical clock. It is to address this problem that the coupling of the two clocks, described previously, is performed. Therefore, the specified computation can be performed exclusively using logical clock values.)

To summarize, before a site W can issue a timestamp (i.e., assign it to a message), it must make sure that the issuance of this timestamp does not violate the Guardian Constraint. Therefore, the Send primitive of the Global Clock Layer, before passing on an outbound message to the lower layers of the Global Time Layer implementation, must first determine the timestamp that will be assigned that message by the Local Clock Layer and check it against the LBClock values. If issuing the timestamp (i.e., sending the message) would violate the Guardian Constraint, then W must wait until it receives some additional messages from its guardians that enable it to increment the LBClock values and thereby allow the timestamp to be legally issued. (In other words, in such a case the

Global Clock Layer will not complete the sending of the message until that later time.) The purpose of the Timesignal message is to avoid forcing a site to wait (arbitrarily long) periods before issuing a new timestamp.

Some additional comments are in order concerning the guardian mechanism. The number of guardians is a system parameter, and represents a tradeoff between cost and reliability. In order to operate correctly, the Global Clock Layer requires that at least one of a site's guardians be UP while the site is down; this argues for a larger number of guardians. However, there is expense (principally in terms of message traffic) associated with an increased number of guardians. It should also be observed that one site can serve as the guardian of several other sites, and that in particular two sites can be each other's guardians.

The Crashsite procedure has been alluded to above as the mechanism employed when a site fails to respond to a message within an anticipated timeout period. The functions of Crashsite are to ensure that the site is really DOWN, and not just slow to respond; to mark the site as DOWN in the Local Status Table; and to increment the local clock in such a way that the timestamps of any messages subsequently locally issued will be greater than the time of the site's failure. (This latter action is required for the simulation of a global clock.)

The implementation of Crashsite has two versions: the first is performed by a site that is a valid guardian of the timed-out site, and the second is used in all other cases. A valid guardian of a site is defined to be a guardian that believes itself to be UP; i.e., a guardian of a site whose own value in its Local Status Table is UP. (Recall that during the first stages of its recovery from a failure, a site is operating but has its own entry in its Local Status Table set to DOWN. This is not switched until a later stage of recovery. Until that point, the site's own local clock is not yet accurate, and so it ought not perform the guardian version of Crashsite, since, as described below, that impacts the clock of the timed-out site.) In the first case, the procedure is as follows:

1. The local clock is incremented by Tdelta. Since the local site (the one performing Crashsite) is a guardian of the timed-out site, the local clock can not be more than Tdelta behind the ward's clock at the time that the latter crashed. Consequently, this incrementation has the effect of pushing the local clock value past the last timestamp issued by the timed-out site.

2. A YOU'RE-DOWN message is sent to the timed-out site, and it is marked as DOWN in the Local Status Table (by means of the MarkDown primitive). If the timed-out site is actually UP, this action will have the effect of crashing the timed-out site while its local clock is still less than than the value just assigned to the guardian's local clock.

It should be observed that these two Steps must be performed atomically; that is, the local clock cannot be accessed or updated by any other process between these Steps. This is necessary to ensure that site status and the clock value are mutually consistent.

If Crashsite is not being invoked at a guardian against one of its wards, then the following procedure is followed:

1. A CrashReport message is sent to some (arbitrarily chosen) guardian of the timed-out site.

2. A response to this message is received. (If the guardian does not respond within the time-out period, then Crashsite is invoked against the guardian, and a new guardian for the original site is selected for Step 1.)

3. The timed-out site is marked as DOWN in the Local Status Table, by means of the MarkDown primitive.

The CrashReport message, when received by a guardian, is processed by its Global Clock Layer. Specifically, the guardian behaves as if it itself had timed-out the ward. The guardian invokes Crashsite (version 1) locally against the timed-out site and then returns a (null but timestamped) response to the CrashReport. The timestamp on the response will be the guardian's local clock value after it completes local execution of Crashsite (which in turn is guaranteed to be greater than the clock value of the timed-out site). Consequently, when the response is received by the site that issued the CrashReport, its clock will be pushed past that of the timed-out site at time of its crash, consistent with the order imposed by a global clock. Furthermore, the secondary execution of Crashsite at the guardian will have had the effect of crashing the timed-out site if it had not really been DOWN.

It should be noted that performing Crashsite does not entail notifying all sites in the network that the site in question has failed. Only the site discovering the fact (and one of the failed site's guardians) need know of the failure. Our approach is that only sites that attempt to interact with a site need know its status, and that each of these can learn of a failure independently. This approach obviates the need for synchronizing the communication of site failure and recovery information among all other sites in the network; this latter issue becomes especially troublesome in the presence of additional failures and recoveries. Instead, each site makes its own determination of the status of other sites. The necessary consistency among these interpretations is provided by means of the global clock. If a site recovers before another one has learned of its failure, then the I'M-UP message (see below) and its processing will assure that any assumptions made about the failed site's clock are universally upheld.

3.5. The Global Status Layer

The Global Status Layer is the topmost level of the Global Time Layer implementation. As such, it is responsible for coordinating the various facilities provided by the layers beneath it and presenting a virtual network with the following characteristics: the existence of a single global clock by means of which all events in the system are timestamped and thereby ordered; every site is at any time in one of two states, either UP or DOWN; transitions between these states, called crashes and recoveries, occur instantaneously with respect to the global clock. The interface to the Global Status Layer provides the abilities to send and receive messages that are timestamped by the global clock, to inquire as to the state of any site, and to set a

Watch on any site. These latter two sets of routines return <status,timestamp> pairs, which indicate that when the Network Clock was at the time indicated by the timestamp, then the status of the site was that specified. This capability is implemented so that the timestamp returned is a current timestamp, rather than one which is obsolete and consequently of little interest, and are demanded in this form by the SDD-1 concurrency control mechanisms.

As has been mentioned, the Global Time Layer provides the view that any site is either UP or DOWN at any time. In reality, of course, this is a fiction. Rather, the behavior of a site in a distributed system can be characterized by one of the following four states:

1. DOWN: i.e., not operating.

2. SLOW: Running, but slow to respond to messages; i.e., not acknowledging messages within a pre-specified time-out interval.

3. FAULTY: Running, responding to messages within the time-out interval, but operating incorrectly; i.e., producting erroneous messages.

4. UP: Running correctly, and responding to messages within the time-out interval.

It is impossible to accurately distinguish among all these four possibilities. A foreign site cannot determine whether another site is failing to respond because it is DOWN or merely because it is SLOW. Consequently, the Global Time Layer merges these two cases. Any site that fails to respond to a message that demands a response within a time-out interval is assumed to be DOWN; however, the system recognizes the possibility that it may merely be SLOW and takes appropriate action (in the form of YOU'RE-DOWN messages). On the other hand, it is not possible for a message handler, such as the RelNet, to distinguish between a site's being FAULTY and its being UP. Consequently, the Local Status Layer only records sites as being UP or DOWN, and the Global Status Layer maintains this views. (However, the RelNet does incorporate the capability for components outside the RelNet to explicitly crash a site that is believed to be FAULTY. This is accomplished by exporting the Crashsite procedure.)

We observe that the Local Status Table and the Global Clock together realize the property that if the site A is marked DOWN in B's Local Status Table, then A crashed before its local clock reached the value of B's clock when it finds the DOWN entry in the table. This fact is exploited by the routine that handles inquiries into the status of a site. The operation of this routine is as follows:

1. Read the value of the Global Clock into t.

2. Examine the entry in the Local Status Table for the site in question.

3. If it is listed as DOWN, then return <DOWN,t>; this indicates that the site crashed before time t (and that it will recover after time t). Note that t is a "current timestamp", since it has just been returned by a read of of the global clock.

4. If the site is listed as UP, then it is still possible that the site is not really UP, that it failed at a time prior to that returned by the global clock and that this fact has simply not yet been recorded in the Local Status Table. To clarify this, a Probe is sent to the specified site.

   a. If a response to the Probe is received within the specified time-out period, then the site is indeed known to be up at time t, and the pair <UP,t> can be returned.

   b. If no response is received, then it must be presumed that the site has failed. Then the Crashsite procedure is called, as it is whenever a site fails to respond within a timeout period. Control then transfers to step 1 of this procedure. The purpose of so doing is to get a new clock value at which to state that the site is DOWN.

The first two steps of this procedure must be performed atomically; that is, no changes to the Local Status Table can be allowed between the time the Global Clock is read and the Local Status Table is inspected. (Again, this is to ensure that the site status and clock value are mutually consistent.)

A user process may request the Global Status Layer to institute a Watch (of either failure or recovery varieties) on a designated site. The Watch primitive at this level will call the Watch primitive provided by the Local Status Layer, to set the table entry. If the user has requested a failure watch (notification when a site crashes), then the Global Status Layer will then periodically issue Probe messages to the site being watched. If the site responds, the watch continues; if it fails to respond within a time-out period, then it is assumed to have failed, Crashsite is invoked against it, and the caller is informed of the failure (together with a relevant timestamp, as specified above).

When a site recovers, the Global Status Layer is responsible for bringing it back to full operational status. It accomplishes this by sending I'M-UP messages to all other sites in the network, and then waiting for responses. Each such response is timestamped with the local clock value at the responding site. (Each time one of these responses is received, the Local Status Layer will automatically push the local clock past the timestamp on the response. This will have the effect of pushing the clock of the recovering site past any time that another site may have thought that it was DOWN.) If some site does not respond to the I'M-UP message, then the recovering site invokes Crashsite against that site; this will also have the effect of pushing the local clock of the recovering site past the clock value of the (presumably) failed site. After each site has responded or been crashed, the recovering site is ready to resume operation; it does so by calling MarkUp to set its own value in the Local Status Table to be UP and then transferring control to an appropriate location.

By symmetry, the Global Status Layer processes I'M-UP messages received from other sites. It does so by calling MarkUp on the issuing site to cause it to be marked as UP in the Local Status Table; it then issues a response to the recovering site. Note that the change to the Local Status Table will cause any processes that had issued a Watch against the recovering site to be interrupted and notified of its change in status.

3.6. Summary

The Global Status level is the topmost stratum in the implementation of the Global Time Layer. Its primitives, plus some of those of the lower levels, constitute the facilities that the Global Time Layer provides to other parts of the RelNet and to SDD-1 user processes. Specifically, these include a Send primitive, which timestamps each message with the current value of the Global Clock; an associated Receive primitive; a primitive that returns the current status of any site in the network together with a value of the Global Clock at which that status is valid; and the Watch facilities, which notify the user when a designated site changes its status. All of these features are provided in the context of a consistent Global Clock, which accurately models the relative ordering of events occurring at different sites in the network. Some of these facilities will be employed in the remaining sections of this paper, while others are needed by the SDD-1 concurrency control mechanisms.

3.7. Catastrophe

The principal catastrophe situation that can befall the Global Time Layer is brought about by the failure of all of a site's guardians while the site is down. If this results, then other sites in the network will be unable to ascertain the value of the failed site's local clock at the time that it failed; this in turn prevents the accurate simulation of a global clock mechanism. This catastrophe can be detected from within the RelNet, because a site performing Crashsite against the site in question will find itself unable to communicate with any of that site's guardians. However, it would be unsafe for the system to proceed in the face of this catastrophe. By neglecting to accurately synchronize with the failed site's clock, the system would fail to uphold the global clock; in other words, the clock would fail to correctly model the sequence of events in the system. The result might be an inconsistent database, whose contents do not represent the result of a legitimate sequence of database operations. In such an eventuality, a human system administrator manually sets the clocks so as to avoid conflict with the failed site.

4. GUARANTEED DELIVERY

4.1. Introduction

It was observed in the discussion of the Message Transmission Layer that its facilities make no guarantee that a message sent will eventually be delivered to the destination site. The reason for this is that the intended recipient may fail before the message can be delivered. This situation is unacceptable for the SDD-1 context, since in order to insure database consistency, a transaction updating a distributed database must be sure that certain messages (such as UPDATEs [Bernstein et al]) will be eventually delivered to all destination sites. SDD-1 demands a facility by which messages may be designated for "guaranteed delivery". Such messages would be assured of reaching their destination site irrespective of the current or future status of either the sender or receiver. This facility is provided by the Guaranteed Delivery Layer of the RelNet.

Specifically, the Guaranteed Delivery Layer affords the following functionality. Primitives are provided for sending and receiving messages. A sender may designate a message for "guaranteed delivery". In this case, the RelNet guarantees that, if the destination site is currently down, it will receive the message upon its recovery. More precisely, the RelNet guarantees that if a sender sends two messages to a receiver, where the first is marked for guaranteed delivery, then the receiver will receive the first before the second. Note that the RelNet can not guarantee that any message (even one marked for guaranteed delivery) is certain to be received, since the destination may never recover from its failure.

The Guaranteed Delivery Layer will provide the sender of a guaranteed message with a subsequent acknowledgment (via an interrupt, for example) when the message has been processed by the RelNet for eventual delivery to the destination. (This will entail making an appropriate number of copies of the message and storing them within the RelNet; this mechanism is detailed below.) Once it has received this acknowledgment, the sender can be certain the message will reach the destination, should the destination eventually recover from its failure. Upon receipt of a guaranteed message by the destination, the receiving process must provide a further acknowledgement to the Guaranteed Delivery Layer. This acknowledgement of receipt is a guarantee by the receiving process that the message has or will be acted upon. (Typically this requires that the message has already been processed and its effects secured on stable storage or that the receiving process has placed the message on stable storage for future processing.) Only after this acknowledgement of receipt has been issued to the RelNet will the Guaranteed Delivery Layer consider the message to have been delivered. I.e., if the destination site fails before issuing this acknowledgment, the Guaranteed Delivery Layer will again provide it with the message upon its subsequent recovery.

It should be noted that the Guaranteed Delivery Layer accepts for sending both guaranteed and non-guaranteed messages. Although no special handling is performed for the non-guaranteed messages, their proper sequencing with respect to guaranteed messages is assured. Thus, between any sender-receiver pair, messages (guaranteed and non-guaranteed) will be received in the order sent. Non-guaranteed messages, however, may be lost; such losses occur during periods when the receiving site is down.

A further capability provided by the Guaranteed Delivery Layer is the Check primitive; this enables a site to determine if it has received all messages sent to it by another site before a given time. Thus a call on this primitive has two arguments, a site and a timestamp. A positive response is returned to the caller if all messages from the given site sent prior to the given timestamp have been received; that is, the response is positive if the next message received from that site is certain to have a higher timestamp than that given as an argument. Otherwise, the response is negative. This capability is employed in SDD-1 by a recovering site to ensure that it has received all messages that were sent to it while it was down, and to ensure that all messages sent by a failed site before it crashed have been received [Bernstein et al]. The details of the implementation of this facility are straightforward and will not be discussed in this paper.

The guaranteed delivery of messages is accomplished by the user of a mechanism we call a Reliable Buffer. There is one such buffer for each destination site. When a user flags a message to a down site for guaranteed delivery, the RelNet establishes the message in that site's buffer; having done so, it returns an acknowledgement to the sending process, since (assuming the RelNet does not fail) the message will now be available for retrieval when the destination site recovers. During the recovery process of a failed site, it will request the RelNet to provide it with all messages in its Reliable Buffer. The recovering recipient will then establish these messages on its own stable storage; and upon completion of its recovery process, the site will process these messages as though they appeared normally in its input queue.

The Reliable Buffer is a mechanism internal to the RelNet; it is implemented by means of the appropriate replication and coordination of each message at several sites in the network. This approach differs significantly from a technique that has been called "persistent communication" (see, e.g., [Alsberg and Day]). In a persistent communication strategy, a message to be delivered to a crashed site is buffered only at the sending site. In such a situation, if the sender is down when the recipient site recovers, the message cannot be forwarded to the recipient. The assumption of a persistent communication scheme is that, at some point in the future, both the sending and the receiving sites will simultaneously be up and the message can be delivered at that time. This assumption is unsatisfactory for the SDD-1 environment. SDD-1 demands that a recovering site be immediately able to retrieve all messages sent to it while it was down; persistent communication does not provide this capability, because the sender might be down when the recipient recovers.

This capability to retrieve all messages sent prior to a given timestamp is needed for the efficient implementation of the SDD-1 concurrency control protocols. Under the SDD-1 concurrency control strategy, it is necessary, in synchronizing against a crashed site, to obtain all UPDATE messages sent by it before it failed. If these messages could not be immediately obtained, then it would be necessary for the concurrency control mechanisms to wait until such time as the messages could be obtained (see [Bernstein et al.]) Under a "persistent delivery" strategy as outlined above, this would mean that the synchronizing site would have to wait for the recovery of the site against which it was synchronizing. In the design of the SDD-1 reliability mechanisms, we have avoided any approach that might require one site to wait for others to recover, because such recovery might be arbitrarily delayed.

The problem with which we are faced is to design a Reliable Buffer that will be available in spite of site crashes. The presentation of a design for accomplishing this is the main topic of this section. An outline of the remainder of this section follows:

1. We introduce the basic implementation of the Reliable Buffer. For purposes of robustness, the Reliable Buffer is replicated at a number of different sites, each replication being called a spooler.

2. We discuss alternative strategies for recovering messages from the spoolers and for switching them cleanly from a spooling to a non-spooling mode. We present the details of the particular strategy that is used, under the simplifying assumption that no spooler crashes during the message recovery process.

3. We next consider the possibility of spooler crashes at three critical points in the algorithm.

   a. First, we consider the case in which the spooler crashes while messages are being removed from the spooler by the recovering site. In this case, the recovering site simply switches to a different spooler. However, the new spooler may have a different message ordering than the original spooler. The notion of an <u>acknowledgement vector</u> is introduced to deal with this problem.

   b. We then consider the possibility that a spooler site crashes while messages are being inserted into it by sending sites. We argue that the spooler site can recover by inserting a <u>gap marker</u> into its message stream to indicate a point at which it may be missing messages.

   c. Third, we consider the possibility that the spooler crashes during the transition from spooling to non-spooling mode, and introduce mechanisms to eliminate the undesirable effects this may have.

4. Finally, we consider the possibility that a recovering site crashes while it is removing messages from a spooler. We show that no harmful consequences result from this, so long as the acknowledgement vector is maintained on stable storage. Furthermore, the mechanism operates correctly when spoolers as well as the recovering site crash.

4.2. Reliable Buffer Implemented As Multiple Spoolers

As mentioned above, there is a Reliable Buffer associated with each site, whose function it is to hold messages sent to it while it was down. A Reliable Buffer is implemented as a set of physical buffers located at several sites in the network. These buffers are known as spoolers. Associated with each site is a set of spooler sites at which the Reliable Buffer for that site is implemented. A single spooler would, in general, be inadequate since it would be susceptible to crashing itself. To protect against this occurrence, a number of spoolers are used for each site. In the sequel, we will assume that while the destination site is DOWN, at least one spooler is always UP. A RelNet catastrophe occurs if this is not the case.

The basic strategy for spooler implementation is as follows. If a sender wishes to reliably buffer a message, the RelNet will send a copy of that message to all spoolers associated with the destination site. When all the spoolers have acknowledged receipt, the message is considered to be reliably buffered. When the recipient recovers, it issues a request to any one of its spoolers to obtain its buffered messages.

One might be tempted to believe that each spooler holds an identical copy of the (logical) Reliable Buffer. However, this need not be true, since different spoolers may contain the same messages in different orders. Consider the case of two spoolers S1 and S2 associated with a destination site C, and two sending sites, A and B. The following sequence of events may occur:

1. A sends message M1 to S1 and receives acknowledgement of its receipt.

2. B sends message M2 to S2 and receives acknowledgement of its receipt.

3. A sends message M1 to S2 and receives acknowledgement of its receipt.

4. B sends message B2 to S1 and receives acknowledgement of its receipt.

In this case S1 has M1 preceding M2, while in S2, M2 precedes M1. However, both message orderings are "correct".

4.3. Implementation Alternatives

While the general strategy of using spoolers is not conceptually difficult, the details of the implementation may become quite intricate. Furthermore, a number of different implementation strategies are possible. We can outline three general approaches, differing in the way in which new messages destined for a recovering site are handled while the spoolers are being emptied by that site. The three approaches are as follows:

Strategy 1: Prior to emptying the spoolers, the recovering site sends a message to all sending sites indicating that they are to cease sending messages to it, either indirectly (via spoolers) or directly. Any messages destined to the recovering site are to be held at the sender. After the spoolers have been emptied, a further message is sent from the recovering site to the sending sites directing them to henceforth send messages directly to the recovering site. In particular, any pending messages being held at the sender may now be sent, and will then be acknowledged on receipt.

Strategy 2: Rather than holding their messages while the spoolers are being emptied, the sending sites continue to send messages to the spoolers. Eventually the spoolers will be emptied, after which point new messages will be sent directly to the recovered site. (Since we can safely assume a receiver can receive messages faster than the senders are sending them, the spoolers <u>will</u> eventually be emptied.)

Strategy 3: While the spoolers are being emptied, new messages are sent directly to the recovering site where they are kept in a temporary local buffer. After emptying the spoolers, the recovering site empties this local buffer before receiving any further messages directly from senders.

The strategies have been presented in order of increasing efficiency and complexity. Strategy 1 is simple to implement but suffers from the disadvantage that new messages cannot be acknowledged until the spoolers have been emptied by the recovering site. Since spooler emptying may be a lengthy process, this strategy was not deemed to be acceptable. Strategy 2 overcomes this problem, but suffers from the disadvantage that, even though the crashed site has recovered, messages to be sent to it must make two "hops" through the network, one to the spoolers and another from the spoolers to the recovering site. In strategy 3, only one "hop" is needed, but the details of implementation are quite complex. Particularly troublesome are the problems raised should the recovering site crash while it is in the midst of emptying its spoolers. When it subsequently recovers, there will be both old and new messages in the spoolers; the old ones must precede, and the new ones must follow, the messages that had been placed in the temporary local buffer at the recovering site during its previous recoveries. Further, these old temporarily buffered messages must be kept separate from any new temporarily buffered messages that may arrive during recovery. After complete details had been developed for this approach, it was felt its improved performance did not justify the complexity of the resulting software. Consequently, Strategy 2 was selected for use in SDD-1, representing a trade-off between efficiency and simplicity.

4.4. Basic Implementation Algorithm

In this section, we present the basic algorithm employed by the Guaranteed Delivery Layer in implementing its send and receive primitives for messages designated for guaranteed delivery. This implementation employs several of the facilities provided by the Global Time Layer. The algorithm below is cast in terms of a set of senders, a receiver, and a set of spoolers. (The generalization to multiple receivers is immediate.) We assume that the set of spooler sites is known to the Relnet component at the sending site. Each of these sites is in one of two modes (spooling or non-spooling). The collection as a whole is said to be non-spooling if both the sender and the receiver are in non-spooling mode, and is spooling if the sender and the spoolers are in spooling mode and the receiver is either down or spooling. All other combinations of local states correspond to transient or illegal global states. Global state change is effected by sending messages to cause appropriate local state changes. Basically, the sender is responsible for changing the global state to spooling when it discovers that the receiver is down. The spoolers, after they have been emptied, are responsible for changing the global state back to non-spooling mode. In the basic algorithm outlined below, we do not provide for spooler crashes. These will be considered in the following section.

The algorithm is expressed by describing how each site behaves when its local mode is spooling, when its local mode is non-spooling, and upon its recovery from a crash.

1. SENDER

a. Sender in non-spooling mode:

To send a message, send the message directly to the receiver, and await acknowledgement. If the acknowledgement is not received within the time-out period: invoke Crashsite against the receiver; send START-SPOOLING messages to all the spoolers for the receiver; enter spooling mode and send the message as specified below.

b. Sender in spooling mode:

To send a message: send the message to all spoolers, and await acknowledgements. If, instead of an acknowledgement, a STOP-SPOOLING message is received from some spooler, enter non-spooling mode and send the message as specified there.

c. Upon recovery of sender:

enter non-spooling mode.

2. RECEIVER

a. Receiver in non-spooling mode:

Await messages sent directly from senders, acknowledge upon receipt.

b. Receiver in spooling mode:

(If the receiver is in this mode, it is recovering spooled messages.) Upon entering spooling mode, a particular spooler is chosen. A NEXT-MESSAGE-PLEASE message is sent to this spooler. In response, the next message in the spooler will be returned (so long as the spooler is not empty); the message is stored on the site's input queue on stable storage, and acknowledgement of its receipt is sent to the spooler. Repeat until a STOP-SPOOLING message is received from the spooler; at that time, enter non-spooling mode.

c. Upon recovery of receiver:

enter spooling mode.

3. SPOOLER

a. Spooler in non-spooling mode:

Upon receipt of message from a sender: if it is a START-SPOOLING message, enter spooling mode; otherwise, respond with a STOP-SPOOLING message. Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver: send STOP-SPOOLING message to receiver.

b. Spooler in spooling mode:

Upon receipt of message from a sender: add message to buffer and acknowledge.

Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver: send next message in buffer to the receiver, and after it has been acknowledged, delete the message from the queue. When the last message has been deleted from the queue, enter non-spooling mode.

c. Upon recovery of Spooler:

We have assumed spoolers do not crash in this version of the algorithm.

4.5. Spooler Crashes

We will examine the problem of spooler crashing in two contexts. The first case is that of spoolers crashing while they are being emptied by a recovering receiver. The second is that of spoolers crashing while the receiver is still DOWN. (If a spooler crashes at any other time, it can be handled in the same way as in this second case.) After considering these issues we present the complete algorithm including provision for these spooler crashes.

If a spooler crashes while it is being emptied, the recovering receiver should switch to a new spooler. However, there is a complication here: many of the messages in the new spooler may have already been received from the former spooler. The new spooler ought not to have to send these messages. To this end, an acknowledgement vector is maintained by the receiver. This is an array indicating, for every sender site, the timestamp of the last message from that site that the receiver has received and acknowledged. Before emptying a new spooler, the receiver sends an ACKNOWLEDGEMENT-VECTOR message, which contains the receiver's current acknowledgement vector. Upon receipt of the ACKNOWLEDGEMENT-VECTOR message, the spooler uses it to delete all messages in its queue that have already been received by the receiver.

The acknowledgement vector is also used by the receiver to allow it to ignore messages that have been previously received , but are nonetheless received again. This can occur, for example, when a sender crashes after having sent a message to some, but not all, of the spoolers for the recipient. Suppose that the recipient then recovers and obtains the message from one of the spoolers that did get it. When the sender subsequently recovers, the process that initiated the message may resend it to the recipient, since the Guaranteed Delivery Layer only acknowledges the message after it has been established at all spoolers. Thus, the recipient will receive two copies of this message.

We now consider the problem of spoolers that crash while the receiver is DOWN. If the spooler remains DOWN until the receiver has recovered and emptied some other spooler, no problems can arise. However, if a spooler does crash and subsequently recovers while the receiver is still DOWN, that spooler's message queue will reflect a "gap" during which it received no messages. If the receiver, upon its recovery, chooses to empty this spooler, then the receiver will not receive those messages sent while the spooler in question was down.

One simple solution to this problem would be to disallow the receiver from emptying spoolers that had crashed and recovered in this manner. The difficulty with this approach is that it will unnecessarily result in a catastrophe in those cases where every spooler has crashed at least once during the period that the receiver was down; even though every message sent to the receiver may be available in some buffer, no spooler would have them all. This would mean, under this approach, that at least one spooler must remain up during the entire period that the receiver is DOWN. This is an unreasonable expectation considering the fact that sites may be DOWN for very long periods of time. Instead, it would be better to have spoolers (on recovery) mark the gaps in their queues during which they were DOWN, and to have the receiver fill those gaps from messages held in other spoolers. Under this strategy, a catastrophe is prevented so long as that collection of spoolers that are up at the time of the receiver's recovery hold all of the messages that had been sent to it while the receiver was down.

To this end, the following conventions are followed: whenever a spooler recovers, it immediately places a GAP marker in its message buffer. This indicates the point at which messages may have been lost. Secondly, each sender remembers the timestamp of the last message it has sent to the receiver.[1] (This can be accomplished by maintaining at each sender an array, PMT, which contains the previous message timestamp for each receiver. PMT must be maintained on stable storage.) When a

[1] This message need not yet have been acknowledged. The intention is for the sender to be able to inform a recovering spooler of the last message that it may have missed.

sender sends a message to a spooler, it appends to that message the timestamp of the previous message that it sent to the receiver. When unspooling, the spooler now behaves as follows. When the spooler receives a NEXT-MESSAGE-PLEASE message and the next item in the buffer is a GAP marker, a GAP message is sent to the recovering receiver. The GAP marker remains in the message buffer. Upon receipt of such a GAP message, the receiver chooses another spooler from which to obtain the remainder of its messages. (The first step in this process, as described above, is to send the new spooler an ACKNOWLEDGEMENT-VECTOR message, which it uses to delete messages already obtained elsewhere by the receiver.) The receiver then retrieves messages from this second spooler until the spooler is emptied, until this spooler crashes or until another GAP is encountered. In the latter two cases, the receiver will move on to another spooler; in particular, it may return to one that it had left earlier upon receiving a GAP message. (The receiver may return to an earlier spooler only after it has received at least one additional message from some other spooler. As usual, the recipient site will reestablish the interaction with the earlier spooler by sending it an ACKNOWLEDGEMENT-VECTOR message.)

In addition to deleting messages that the acknowledgement vector indicates have already been received by the recipient, the receipt of an acknowledgement vector message by a spooler will cause it to delete any GAP markers in its buffer that are no longer operative. Intuitively, a GAP marker is operative if there may be messages for the receiver that are missing from the buffer and whose place is occupied by the GAP marker. The spooler can establish if a GAP marker is operative by examining the acknowledgement vector sent to it by the receiver and the messages in its own queue.

A GAP marker is no longer operative if, for each sender site, there is a message in the buffer following the GAP marker whose predecessor (as indicated by the previous message timestamp that is attached to each message) has already been received by the recipient. After deleting the inoperative GAP markers, the spooler can resume sending spooled messages to the receiver.

We must also consider the situation in which the spooler crashes immediately after having sent a STOP-SPOOLING message to the receiver. A sender in this case may not know that the receiver has entered non-spooling mode and will continue to send messages to the spoolers that are still UP (who also have not learned of the state change). To prevent this situation, the receiver, after receiving a STOP-SPOOLING message from one spooler, does not immediately enter non-spooling mode. Instead, it switches to another spooler in the usual way and attempts to retrieve messages from it. (In most cases, the second spooler will not have any additional messages, but in the situation described above,where the first spooler crashed before notifying senders of the state change, there may indeed be some new messages there.) The receiver then enters non-spooling mode only after having received a STOP-SPOOLING message from all of the UP spoolers.

Finally, we must consider how the sender will deal with failing and recovering spoolers. The sender will crash any spooler that does not acknowledge receipt of a message sent to it; this will ensure that the spooling process is accurately begun and that messages are securely spooled. When a failed spooler recovers, the sender brings it into the spooling process by issuing it a START-SPOOLING message.

4.6. Crash of the Recovering Receiver

The recipient may crash while it is in the process of unspooling. Upon its recovery, it simply chooses a spooler and resumes the unspooling operation. It should be noted that the acknowledgement vector must be the same as at the time of the recipients' crash in order for messages not to be received twice. This requires that the recipient keep its acknowledgement vector (or more precisely, the information that it contains) on stable storage. If the recipient maintains its input queue on stable storage, then the information needed to reconstruct the acknowledgement vector is available from this input queue.

4.7. Complete Algorithm For Reliable Buffer Implementation

The complete algorithm is summarized below:

1. SENDER

   a. Sender in non-spooling mode:

   To send a message, send message directly to receiver and await acknowledgement. If the acknowledgement times-out: invoke Crashsite against the receiver and send START-SPOOLING messages to all the spoolers for that receiver. Enter spooling mode and send the message as specified below.

   b. Sender in spooling mode:

   Upon entering spooling mode, establish recovery watches against those spoolers that are DOWN. To send a message, append to message the current value of PMT [receiver] and then set PMT [receiver] to the current global clock time.

Send message to all spoolers that are currently UP, and await acknowledgements. If, instead of acknowledgement, a STOP-SPOOLING message is received from some spooler, cancel the recovery watches against DOWN spoolers, enter non-spooling mode, and send the message as specified there. If an acknowledgement times-out, invoke Crashsite against that spooler and establish a recovery watch against it.

If a crashed spooler recovers: Send START-SPOOLING message to the spooler.

c. Upon recovery of sender:

Enter non-spooling mode.

2. RECEIVER:

a. *Receiver in non-spooling mode:*

Await messages sent directly from senders; upon receipt, update acknowledgement-vector and acknowledge the message.

b. Receiver in spooling mode:

Upon entering spooling mode, a particular UP spooler is chosen. An ACKNOWLEDGEMENT VECTOR message containing the current acknowledgement vector is sent to the spooler. (A response is expected; if none is received within a time-out period, Crashsite is invoked against the spooler and another is selected.) Successive NEXT-MESSAGE-PLEASE messages are then sent to retrieve messages from the spooler. If the spooler does not respond to a NEXT-MESSAGE-PLEASE within a time-out period, Crashsite is invoked against it and another spooler is selected for unspooling. If the spooler replies with a regular message, update the acknowledgement vector, acknowledge the message, establish the message on the input queue on table storage and acknowledge it. If the spooler replies with a GAP message or a STOP-SPOOLING message, initiate unspooling from another spooler. The unspooling procedure terminates when a STOP-SPOOLING message has been received from all UP spoolers, at which point non-spooling mode is entered.

c. Upon recovery of receiver:

Reconstruct the acknowledgement vector if necessary; enter spooling mode.

3. SPOOLER

a. Spooler in non-spooling mode:

Upon receipt of message from a sender: if a START-SPOOLING message, acknowledge and enter spooling mode. Otherwise, respond with a STOP-SPOOLING message.

Upon receipt of NEXT-MESSAGE-PLEASE or ACKNOWLEDGEMENT-VECTOR message from the receiver: send STOP-SPOOLING message to the receiver.

b. Spooler in spooling mode:

Upon receipt of message from a sender: add message to the queue (on stable storage) and acknowledge.

Upon receipt of ACKNOWLEDGEMENT-VECTOR message from the receiver: delete all previously received messages from the queue as well as GAP markers that are no longer operative and acknowledge. If the queue becomes empty, enter non-spooling mode.

Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver:

If the next item in the queue is a GAP marker, send a GAP message. If the next item in the queue is a message, strip the previous message timestamp from it and forward it to the receiver, and remove the message from the queue when the receiver has acknowledged it. If the buffer becomes empty, enter non-spooling mode.

c. Upon recovery of Spooler:

Place GAP marker in the message buffer.

4.8. Spooler Catastrophe

The algorithm as just described requires that at least one spooler be UP whenever the receiver is down. When this condition is not met, a spooling catastrophe may occur. . This catastrophe is detected by the sender when no UP spoolers are available for spooling. The catastrophe is detected by the receiver, when all spoolers that are UP return GAP messages in response to a NEXT-MESSAGE-PLEASE message. By providing for additional or more robust spoolers, the likelihood of spooler catastrophe can be decreased.

5. THE TRANSACTION CONTROL LAYER

5.1. The Atomicity of Transactions

In this section, we describe how the Relnet supports the atomic execution of distributed database transactions, which access and modify data items that may be stored (and replicated) at several sites in the network. Reliability mechanisms are needed to ensure the correct execution of these transactions despite asynchronous site failures (in particular, these that occur during the execution of a transaction).

Following [Eswaran et al.], we define a transaction as an atomic database operation at the user level. That is, the user is given the ability of grouping together a number of primitive database operations and designating the group to be a transaction; the system must then behave as if each such transaction is processed as an atomic, indivisible, unit. A transaction is specified to the system in terms of a sequence of actions, each action being an atomic operation at the system level. Even though execution of actions from different transactions may be interleaved by the system, it must preserve the outward appearance of having executed one transaction to completion before beginning another.

The atomicity constraint guarantees, for example, that it is not possible for any transaction to read data that has been partially, but not completely, updated by another transaction. Nor is it possible for two or more transactions to interleave their read and write operations so as to result in "reader-writer" anomalies. Such an anomaly could result from a scenario in which one transaction, T1, reads a variable, x; another transaction, T2, then reads the variable x; next transaction T2 updates x; and finally T1 updates x. (Consider what happens when both T1 and T2 both increment x by 1.) The difficulty arises because T1's update is based on a data value that has become invalidated by T2's update.

It is the responsibility of the concurrency control mechanism of SDD-1 to guarantee the atomicity of transactions. Atomicity is achieved by forcing read operations to wait until appropriate update operations have completed before they are allowed to execute. (Discussion of the SDD-1 concurrency control techniques is given in [Bernstein et al.] and a general survey of distributed concurrency control techniques can be found in [Bernstein and Goodman].)

One may reasonably ask why reliability mechanisms are necessary for transaction atomicity if the concurrency control strategy is correct. There are two reasons. First, the concurrency control algorithms themselves must be made safe against site failures. Second, the execution of a transaction will typically entail the sending of update messages from one site to a number of other sites; and even with properly functioning concurrency control, the sending site may fail before having issued all of the update messages associated with some transaction. A situation in which some, but not all, of the update messages associated with a transaction have been received and processed results in database inconsistency and negates the principle of atomicity. Although the unsent messages, which are "buried" at the failed site, could presumably be issued upon the node's recovery, any read operations waiting on the completion of that transaction would they be forced to wait until such time as the recovery actually occurred.[1] Such delays would be intolerable. It is a general principle of SDD-1 that a transaction should

[1] Assuming that the site has not failed permanently, in which case the reading transaction would have to wait forever!

never be forced to wait for a site to recover in order to run to completion.[1]

The first issue identified above, that of failure-proofing the concurrency control mechanism, is dealt with elsewhere ([Bernstein et al.]) and will not be covered here. The solution to that problem is based on the judicious use of RelNet capabilities.

The second issue, the problem of unsent updates at a failed site, is the central focus of this section. This problem can also be conceived of as that of "atomic message broadcast", enabling a site to send a collection of messages to different sites as a single package (i.e., eliminating the possibility of the sending site failing partway through the process). The problem is resolved by an _atomic commit_ procedure. This procedure ensures that whenever any update messages become buried, the transaction with which they are associated will be _aborted_; i.e. _none_ of its updates will be performed. Only when _all_ of the update messages for a transaction have been issued will the transaction be _committed_; at that point, _all_ of the updates will be guaranteed to transpire. Thus it will not be necessary for any read operation to wait for a site to recover before proceeding. In those cases where a read might have been forced to wait for a buried update, the updating transaction will be aborted, hence obviating the need for the read to continue waiting.

[1] Except, of course, in the case where the transaction seeks to read data that is stored at the failed site and nowhere else in the network.

It is important to distinguish the intent of our atomic commit operation from similar mechanisms proposed in other contexts. For example, the two-phase commit operation of [Gray], upon which our atomic commit is based, is designed for systems in which an update message may be rejected (for example, as a result of concurrency control considerations). In such a scheme, any update message rejection mandates the abortion of the entire transaction. The two-phase commit protocol insures that no update takes place until all update messages have been accepted. However, this mechanism as well as other proposed variants of it (e.g. [Lampson and Sturgis], [Reed]), may withhold the commit/abort decision until after a failed site has recovered (and given the opportunity to reject the update message). Thus, a site may hold uncommitted update messages for long periods of time. As discussed above, this would not be acceptable in the SDD-1 environment, where other transactions may be waiting for commitment of the update messages.

5.2. Transaction Control Functionality

The SDD-1 mechanisms for transaction control are oriented towards a particular model of transaction structure and operation. In this model, a transaction is invoked by user command at a given site. (Each transaction is assigned a unique identifier.) The invocation creates a process at that site, which is the controlling process for the transaction. This process may then cause the creation of other (cohort) processes (at this or at other sites); the transaction will be realized by coordinated activity of the set of processes. (A process is uniquely associated with a transaction, and is identified by a <transaction identifier, process identifier> pair.) The processes associated with a transaction may communicate with one another and perform local computations. At the end of the transaction, the controlling process sends out update messages, indicating to each site the changes that are to be made to certain database elements stored at the site. The update identifies the data elements and their new values. After this action is completed, the controller successfully terminates (commits) the transaction, causing the updates to take effect. At any point during its execution, the controller may abort the transaction; this may be caused by the failure of a site running one of the cohorts. Until the transaction is committed, it has no effects that can be seen by other transactions.

In order to support such atomic transactions, the RelNet provides the following capabilities:

1. The capability to obtain a new transaction identifier; the requesting process will be the controlling process for the transaction.

2. The capability to create a cohort process at the local or a foreign site. The new process will be associated with the same transaction as the requesting process, and will be destroyed when the transaction is committed or aborted.

3. The capability to request that a transaction be committed or aborted. Both types of request signal the completion of the transaction and may be executed only by the transaction's controlling process. When the transaction is aborted, any updates that may have already been performed by the transaction will be undone.

5.3. The Implementation Environment For Transaction Control

In this section, we will examine the components from which the Transaction Control capability described above will be implemented. As the outermost software layer in the RelNet, the Transaction Control component may utilize the functionality provided by the Reliable Delivery, Status Monitoring and Global Clock components. It addition it relies on a number of facilities assumed to be provided by the local operating system or database management system at each site. These latter capabilities include:

1. The capability to create a new process and associate that process with a given transaction number.

2. The capability to delete such processes.

3. The capability to perform local database updates in a tentative mode. Updates made in this mode are not installed until they have been committed.

4. The capability to abort all tentative updates associated with a transaction, restoring the original value to updated data items.

5. The capability to commit all tentative updates associated with a transaction, causing the new data item values to be visible.

It will be noted that we are essentially assuming the existence of a local transaction control capability out of which we are building a distributed transaction control capability.

The implementation of global process control primitives in terms of message communication and the local primitives is straightforward and will not be discussed here. Our primary concern will be with the implementation of the global commit/abort facility in terms of the local commit/abort primitives. The difficult technical problem here is that of uniformly committing a transaction in the presence of local site failures and recoveries.

5.4. Two-Phase Commit

A two-phase commit procedure, similar to the one described in [Gray], forms the core of our atomic commit mechanism. This procedure is described below. Let C be the controlling process of the transaction and U1, U2,... Un be those cohort processes for this transaction that perform local updating on its behalf.

Phase 1a: C issues UPDATE messages to U1 through Un. These cause the local databases to be updated in tentative mode in accordance with the instructions of the UPDATE.

Phase 1b: C waits for Guaranteed Delivery Layer acknowledgement that these UPDATE messages have been delivered or reliably buffered.

Phase 2a: C issues COMMIT messages to U1 through Un. These cause the transaction to be locally committed.

Phase 2b: C waits for Guaranteed Delivery Layer acknowledgement of these COMMIT messages.

It should be noted that, in the RelNet environment, it is not necessary for the UPDATE message to actually reach the destination process. So long as the message is stable within the Guaranteed Delivery component, it is sure to eventually reach its destination. This acknowledgement is sufficient to assure that the UPDATE will reach its destination and be executed there (in tentative mode).

This procedure is, therefore, insensitive to crashes of U1 through Un before or during its execution. It is, however, sensitive to failures of the commit process C. In particular, if C crashes during Phase 1, some or all of the updating processes will have received UPDATE messages in tentative mode for which no COMMIT will be forthcoming. In such a situation we would like to abort the transaction, and discard any UPDATE messages that have been received. More importantly, should C crash during Phase 2, then all processes (including those that have received UPDATEs but no corresponding COMMITs) should commit the transaction, to ensure uniform installation of the updates.

Our solution to this involves the use of a number of commit <u>backup</u> processes; these are processes that can assume responsibility for completing C's activity in the event of its failure. These backup processes are created by C before it issues any UPDATE messages; each backup, when it is created, is informed of the identity of C's cohort process. Then, if C should fail before completion the transaction, one of the backups will take control. If C failed before issuing all the UPDATEs, then the backup will abort the transaction; otherwise it will commit it. In either case, the desired effect is realized by sending COMMIT or ABORT messages to the cohorts, as appropriate. Naturally, proper coordination among the commit backup processes is crucial; a key issue is the selection of a single commit backup process to take over in the event of C's failure. Our technique for backup selection is described in the next section. Following this discussion, we describe the atomic commit procedure incorporating backups.

5.5. Backup Selection Algorithm

In the event of the failure of the committing process C, we wish one, and only one, commit backup process to assume control of transaction completion. The algorithm for accomplishing this assumes an ordering of the commit backup processes, designated in order as B1, B2, ... Bm. Process Bk-1 is refered to as the predecessor of process Bk (for k>0), and process C is taken as the predecessor of process B1. Initially, each of the backup processes is watching for the failure of its predecessor; that is, Bi+1 is assumed to have issued a Watch on Bi. The following conventions are then followed:

1. If a watched backup process is found to be down, then its watching process begins to watch the predecessor of the failed process instead.

2. If process C is found to be down, the backup process that is watching it assumes control of the transaction.

3. If a backup process recovers, it ceases to be a part of the backup mechanism (i.e., it behaves as if it had stayed down).

These rules have the following consequences:

1. If C fails, at most one backup process will assume control. This will be the lowest-numbered backup that has not failed during the procedure. (Note: If all backups have failed at least once since their creation at the time of C's crash, then no take over will occur. This constitutes a catastrophe and will be discussed below.)

2. If a backup process, Bc, fails while it is in control of the transaction, then again at most one backup process will take over. This will be the backup Bk that was watching Bc, at the time of its failure. Having watched Bc fail, Bk will examine each of Bc's predecessors, find them all down, and eventually discover that C itself is down. At that point, Bk assumes control.

Thus, at most one process, either C or one of its backup processes, will be in control at any given time. Having assumed control, a backup will proceed to issue COMMIT or ABORT messages, depending on its state; this issue is addressed in the next section.

5.6. Atomic Commit With Backups

The following four-phase procedure is used to implement atomic commit in the RelNet. Below, C is the process initiating the commit and U1, U2,... Un are its cohort processes that perform local updating on behalf of the transaction. We first describe the procedure as executed by C.

Phase 1a: C establishes m commit backup processes B1, B2,...Bm. When it creates process Bi, C informs it of the identity of all lower-numbered backup processes and of cohort processes of the transaction.

Phase 1b: C waits for an initial message from each of the backup processes to confirm its existence. Crashsite is invoked against any backup site that fails to respond within a time-out interval.

Phase 2a: C issues UPDATE messages to U1 through Un. These will cause the local databases to be updated in tentative mode.

Phase 2b: C waits for Guaranteed Delivery acknowledgement of these UPDATE messages.

Phase 3a: C issues COMMIT messages to all B1 through Bm.

Phase 3b: C waits for each backup to acknowledge receipt of the COMMIT. Crashsite is invoked against any backup that does not acknowledge within the time-out period.

Phase 4a: C issues COMMIT messages to U1 through Un. These cause the transaction to be locally committed.

Phase 4b: C waits for Guaranteed Delivery acknowledgement of these COMMIT messages, and then destroys the backup processes. This represents the successful completion of the transaction.

Each backup is always in one of two states: the abort state or the commit state. Its state is determined by the following transition rules: when created, a backup process enters the abort state; receipt of a COMMIT message causes it to enter the commit state; receipt of an ABORT message (discussed below) causes it to return to the abort state. The backup process state corresponds exactly to the desired global effect to be achieved. Thus, if a backup process assumes control when it is in the abort state, it should send ABORT messages to all of U1 through Un; if it is in the commit state, it should send COMMIT messages instead.

The operation of a backup process can then be described as follows:

1. When created, a backup sends a message to C, confirming its creation. It also issues a failure watch against its predecessor.

2. Upon receipt of a COMMIT or ABORT message, it acknowledges and makes the appropriate state transition.

3. When notified of the failure of the process it is watching:

   a. If it is watching C, the backup assumes control of the transaction. It issues COMMITs or ABORTs to all the cohorts, depending on its state.

   b. If it is watching another backup, it issues a failure watch against that backup's predecessor.

4. If the site on which the backup runs should crash, then the backup ceases to participate in this activity. I.e., the backup process is not continued upon site recovery.

The foregoing works correctly so long as the backup that assumes control does not crash in the midst of sending the COMMIT or ABORT messages to U1, ... Un. If it were to so crash, there is a danger that the next backup to assume control would be in a different state than the failing backup. In such a case, some update process might receive both a COMMIT and an ABORT message. To avoid this problem, each backup process, when created, will be informed of the identity of its higher-numbered backups, which represent the sites that might assume control from it. Then before performing any backup activity, a backup process will ensure that these higher-numbered backups are in the same state that it is in.

Specifically, upon assuming control, a backup process, B, performs the following two-phase procedure (replacing 3a above):

Phase B1a: B issues COMMIT or ABORT messages (depending on its current state) to all higher-numbered backup processes.

Phase B1b: B waits for each of these backups to acknowledge receipt of the message. Crashsite is called against any backup that does not acknowledge within the time-out period. (This is to ensure that any backup that is UP has received the message.)

Phase B2a: B issues COMMIT or ABORT messages (depending on its current state) to all of U1 through Un.

Phase B2b: B waits for Guaranteed Delivery acknowledgement of these messages, and then destroys all the backup processes.

By following this procedure, B insures that, before it ever sends out any messages to the updating processes, all remaining backup processes are in the same state that it is in. Therefore, it will not be possible for an updating site to receive both a COMMIT and an ABORT issued by different backup.

5.7. Catastrophe In Commit

The algorithm presented in the preceding section will succeed so long as at least one member of the set of the sites {C, B1,....,Bn} remains UP throughout the 4-phase procedure. If this condition does not hold, a commit catastrophe occurs. This catastrophe is not automatically detected by the Relnet. The effect of the catastrophe, however, is simply that some or all of the updates will not be installed. This may force other transactions to wait indefinitely for the completion of the suspended transaction, but it will not produce an inconsistent database. It is left as the responsibility of a system administrator to detect the commit catastrophe and manually issue COMMIT or ABORT messages as appropriate. The likelihood of a commit catastrophe can be lessened by using additional or more stable backup commit sites.

6. CONCLUSION

In this paper, we have set forth the basic functionality and architecture of the RelNet, and described implementation mechanisms for its most important facilities. A number of the individual algorithms (such as those for guaranteed delivery and transaction control) should be transportable to other contexts. The Global Time Layer, on the other hand, describes a facility, based on a global clock, which may be specific to the needs of SDD-1. However, we believe that this notion of a uniform clock as a means for coordinating a distributed activity possesses numerous attractive features and may serve as the basis for further developments in distributed system design.

An implementation of the RelNet is currently underway at Computer Corporation of America; it is expected to be operational by 1980. Needless to say, we could not specify the complete details of such an implementation effort in this paper. Numerous challenging problems are being confronted and solved in that context. We have sought to emphasize the basic principles of the RelNet on which the implementation is based.

Distributed system reliability remains very much an active problem. We have only proposed a first cut at a solution to it. Many of our algorithms need improvement or enhancement; other techniques may be necessary for different environments, that dictate different tradeoffs between reliability and efficiency. However, we believe that our results will form a foundation on which others will be able to build.

7. ACKNOWLEDGEMENTS

We would like to express our gratitude to many employees of Computer Corporation of America for the contributions that they have made to the research described here. In particular, Jim Rothnie provided the leadership for the SDD-1 project as a whole and served as an effective critic of our work; Sunil Sarin and Terry Landers proved invaluable in locating often subtle flaws in earlier versions of some of the algorithms; the technique for commit processing discussed in Section 5 is based on a suggestion of Nat Goodman's. We would also like to recognize the assistance of Phil Bernstein, Steve Fox, and Chris Reeve.

APPENDIX

In this Appendix, we are concerned with the problem of partition catastrophes in the RelNet, i.e., situations in which communication failures have split the underlying network into two or more independent sub-networks. (In particular, each sub-network may contain only a single site; this would occur when a site becomes disconnected from the rest of the network.) Prior to the partition, SDD-1 keeps the entire database internally consistent by means of techniques described in this paper and in [Bernstein et al]. After the partition has split the network, the same techniques can be employed to keep the sub-database contained in each sub-network internally consistent; however, because communication among the sub-networks has been cut off, it is impossible for the data to be consistent across sub-network boundaries. The key problem is how these potential inconsistencies are to be detected and handled. This issue is most relevant when the partition is repaired and the various sub-databases should be recombined into a single database.

It must first be noted that, in general, there is no way for any system (including the RelNet) to reliably distinguish a network partition from the simple failure of a number of sites (although in some cases it can be done). The reason for this is that site failure can in general only be detected by a site failing to respond to messages sent to it, which is also a symptom of a network partition. One mode of operation would be always to interpret such a situation as site failure; i.e., a site unable to communicate with other sites would assume that they are down, while in fact they might be operating on the other side of a partition. When the partition is repaired, and the two sides reconnected, it would be established that these previously made assumptions were false, and that as a result, the database as a whole is in an inconsistent state and some transactions run during the time of the partition were erroneous. At this point, manual intervention would be required to reconcile the sub-databases into a consistent whole; the nature of the intervention would depend heavily on the semantics of the particular application and the database. In some cases, each sub-database could contribute the correct value of certain data items; in others, an arbitrary selection might be adequate; and in others, a more complex reconciliation would be necessary. More seriously, some transactions run during the time of the partition might have been erroneously executed, since they were performed under the fallacious assumption that sites on the other side of the partition were down. Ad hoc rectification of this situation would have to be performed by the system administrator, who might have to notify some users that results returned to them by a transaction were false or that its effect was not in fact incorporated into the database.

However, this approach is unsatisfactory because of the erroneous transactions that it temporarily allows and because it imposes an unreasonable burden on the system administrator. What is needed is a suitable generalization of the RelNet behavior upon the detection of what it believes to be site failures, so that if indeed the situation is one of network partition rather than site failure, these kinds of inconsistencies will be avoided; database reconciliation upon partition repair could then be performed automatically (or nearly so). The basic approach would be, upon detection of what might be a network partition, to forbid certain transactions from running. A number of possible variations on this idea have been proposed and are discussed here. (We note that in many cases this technique may be too restrictive and less constraining approaches may be feasible. However, characterizing these situations and the alternative approaches is a difficult task.) In the sequel, we consider several variations on the basic theme and propose the network partition problem as a topic for further research.

Obviously, it is impossible for a sub-network to run a transaction that requires data not available in that sub-network; there is simply now way to retrieve that data. Consequently, our sole focus is on how to process transactions that request only data available on the local sub-network. If such a transaction updates data items whose only copies are contained in the same sub-network, then there can be no difficulties, because that transaction has no impact on the other sub-network. Problems are raised only by transactions that seek to update data items stored in another sub-network. This may cause difficulties because transactions running in the other sub-network will be unaware of the updates made by this transaction. Consequently, our goal is some means of managing transactions that attempt to update data items contained in another sub-network.

The most drastic approach would simply be to forbid any sub-network from running such transactions while a partition is in effect. However, this is extreme. A less restrictive solution would be to allow only one of the sub-networks to run such transactions while the partition is in effect; the other sub-networks would only be allowed to run transactions that update local data items. After the partition is repaired, the distinguished suo-network will report its updates to the other sub-networks, which can then install them.

The difficulty here is to ensure that one and only one sub-network remains "update-active". Unfortunately, it is not possible for the sub-networks to communicate with one another in order to decide which one will continue updating. A solution is to define the concept of a "majority sub-network"; only the majority sub-network will be allowed to be update-active. The concept of majority is defined in such a way that a sub-network can decide on its own whether or not it is a majority sub-network; furthermore, at most one sub-network can be a majority. A simple majority definition might be that a sub-network is a majority sub-network if and only if it contains over half the sites in the network. More complex definitions may be more desirable. For example, each site could be assigned a weight. Then a sub-network would be a majority if the sum of the weights of its sites were greater than half the sum of the weights of all the sites. A special case of such a majority definition would be, for example, "A sub-network is a majority sub-network if and only if it contains site 13". (In this case, site 13 would be assigned a weight of 1 and all others a weight of 0). The notion of majority sub-network is utilized in the Thomas algorithm for distributed concurrency control [Thomas].

This approach would be implemented as follows. Upon detection of a possible partition, a site would determine whether or not it belonged to the majority sub-network; if it did not it would limit the kinds of transactions it could run.

The majority sub-network solution is undesirable in general because, in a distributed system a data item is typically associated with a single site that uses it heavily, even though other sites may have local copies of it. For example, a warehouse would keep its inventory data locally, although a copy of this data might also be kept at the corporate headquarters site. When a partition splits the warehouse from corporate headquarters, it would be desirable for the warehouse to be able to continue updating its own inventory data, even though copies of it are stored in the other sub-network and the warehouse site itself not be in the majority sub-network. This observation leads to an alternative policy. Each data item is defined to have a "primary copy". A sub-network can then update a data item so long as its primary copy resides in that sub-network. Non-primary copies residing in other sub-networks will be updated when the partition is repaired. In our example, the primary copy of the inventory data would reside at the warehouse site. In event of a partition separating the warehouse from corporate headquarters, the warehouse would be able to update the inventory but the corporate headquarters would not. When the partition is repaired, the headquarters' copy of the inventory would be copied from the other. This is an improvement over the majority sub-network approach, because there the warehouse would be able to update its inventory only if it happened to be in the majority sub-network.

The principal problem with the primary copy policy as stated above is that it doesn't work. Consider the following scenario. The network has been partitioned into two sub-networks, P1 and P2. There are two data items, X and Y. X1 and X2 are two copies of X, Y1 and Y2 are two copies of Y. X1 and Y1 reside in partition P1, while X2 and Y2 reside in P2. The primary copy of X is X1, the primary copy of Y is Y2. There are two transactions, T1 and T2. T1 runs in partition P1 and performs the operation X : = Y+1.

T2 runs in partition P2 and performs the operation Y := X+1. Now, under the primary copy policy as stated above, both T1 and T2 are allowed to run since the primary copy of the data item that they update resides in their sub-network. However, such a concurrent execution would be faulty. Assume that before the partition occurred, both X and Y were 0 (and hence, X1, X2, Y1 and Y2 were 0). After the two transactions T1 and T2 had run, X would have value 1 and Y would have value 1. This result is not correct because it fails to satisfy the serializability criterion for concurrent correctness [Bernstein et al]. Under this criterion, the result of any concurrent execution of transactions must be equivalent to some serial execution of the transactions. In the case of these two transactions, there are only two possible serial execution orders. In the first, T1 precedes T2 in which case the result is X = 1 and Y = 2; in the second, T2 precedes T1 in which case the result is Y = 1 and X = 2. Neither of these corresponds to the result obtained in the scenario, X = 1 and Y = 1. Therefore, the primary copy policy as used there is incorrect.

This admittedly involved and subtle example illustrates the necessity of developing a formal understanding, accompanied by proofs, of the correctness of any concurrency or reliability algorithms. A correct statement of the primary copy policy (without proof) is as follows: Every data item has a primary copy; a sub-network is allowed to run an update transaction only if the primary copies of all the data items the transaction reads or writes reside in the sub-network.

Still, the corrected primary copy policy falls short of the ideal. For any data item, only one sub-network would be allowed to update it. When we take into account the semantis of particular applications, it is possible to devise special algorithms that allow several sub-networks to (correctly) update copies of the same piece of data. Consider the case of an airline reservations system. Copies of seat assignment information may be kept at several sites within the network. Assume a 10 site network that becomes partitioned into a 7 site sub-network and a 3 site sub-network. A policy could state that the 7 site network is allowed to allocate up to 70% of the remaining seats on any flight, while the 3 site network is allowed to allocate up to 30% of the remaining seats. No conflicts would then occur. After the partition is lifted, each sub-network would inform the other of how many seats it had allocated during the partitioned operation. This is much more desirable than allowing only one sub-network to allocate seats during the partition.

Given any particular application it is usually possible to conceive of some policy that would allow updates to be made in several sub-networks at once. It is not, in general, necessary to restrict updates on a data item to only one sub-network. It seems clear however, because of the application specific nature of these solutions, that such policies must be specified by the DBA. Further research is needed to determine general policies that are applicable to large sets of applications, and to formulate and verify them in a precise manner.

REFERENCES

[Alsberg and Day]

Alsberg, P.A., and Day, J.D., "A Principle for Resilient Sharing of Distributed Resources," Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 1976.

[Bernstein et al]

Bernstein, P.A.; Shipman, D.W.; Rothnie, J.B., "Concurrency Control in SDD-1: A System for Distributed Databases; Part I: Description," to appear in ACM Transactions on Database Systems.

[Bernstein and Shipman]

Bernstein, P.A., and Shipman, D.W., "Concurrency Control in SDD-1: A System for Distributed Databases; Part II: Analysis of Correctness," to appear in ACM Transactions on Database Systems.

[Gray]

Gray, J.N., "Notes on Data Base Operating systems," in Operating systems: An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, pp. 393-481.

[Lamport]

Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed Systems," Proceedings of the 1978 IEEE COMPCON Conference, September 1978.

[Lampson and Sturgis]

Lampson, B., Sturgis, H., "Crash Recovery in a Distributed Data Storage System," to appear in Communications of the ACM.

[Lorie]

Lorie, R., "Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems, 2(1), March 1977.

[Montgomery]

Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," MIT Laboratory for Computer Science, LCS Technical Report No. 207, Cambridge, Massachusetts, December 1978.

[Reed]

Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," MIT Laboratory for Computer Science, LCS Technical Report No. 205, Cambridge, Massachusetts, September 1978.

[Rothnie et al]

Rothnie, J.B., Bernstein P.A., Fox, S.A., Goodman, N., Hammer, M., Landers, T.A., Shipman, D.W., Reeve, C.L., Wong, E., "SDD-1: A System for Distributed Databases," to appear in ACM Transactions on Database Systems.

[Schapiro and Millstein]

Schapiro, R.M., Millstein, R.E., "Reliability and Fault Recovery in Distributed Processing," Oceans 77 Conference Record, Vol. II, Los Angeles, California, October 1977.

[Schaprio and Millstein]

Schapiro, R.M., Millstein, R.E., "Failure Recovery in a Distributed Database System," Proceedings of the 1978 IEEE COMPCON Conference, September 1978.

[Svobodova]

Svobodova, L., "Reliability Issues in Distributed Information Processing Systems," Proceedings of the Ninth IEEE Fault Tolerant Computing Symposium, Madison, Wisconsin, June 1979.

QUERY
PROCESSING IN SDD-1:
A SYSTEM FOR
DISTRIBUTED DATABASES

Nathan Goodman
Philip A. Bernstein
Eugene Wong
Christopher L. Reeve
James B. Rothnie

October 1, 1979

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. N00039-77-C-0074, ARPA Order No. 3175-6. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Abstract

This paper describes the techniques used to optimize relational queries in the SDD-1 distributed database system. Queries are submitted to SDD-1 in a high-level procedural language called Datalanguage. Optimization begins by translating each Datalanguage query into a relational calculus form called an envelope, which is essentially an aggregate-free QUEL query. This paper is primarily concerned with the optimization of envelopes.

Envelopes are processed in two phases. The first phase executes relational operations at various sites of the distributed database in order to delimit a subset of the database that contains all data relevant to the envelope. This subset is called a reduction of the database. The second phase transmits the reduction to one designated site, and the query is executed locally at that site.

The critical optimization problem is to perform the reduction phase efficiently. Success depends on designing a good repertoire of operators to use during this phase, and an effective algorithm for deciding which of these operators to use in processing a given envelope against a given database. The principal reduction operator that we employ is called semi-join. In this paper we define the semi-join operator, explain why semi-join is an effective reduction operator, and present an algorithm that constructs a cost effective program of semi-joins given an envelope and a database.

Table of Contents

Table of Figures

1. Introduction

SDD-1 is a prototype distributed database management system being developed by Computer Corporation of America. SDD-1 permits a database to be distributed among the sites of a computer network, yet accessed as if it were stored at a single site. Users interact with SDD-1 by submitting transactions written in a high-level procedural language called Datalanguage [CCA]. Query processing in SDD-1 amounts to translating each transaction into a sequence of commands that access data at local sites and move data between sites to perform the transaction's computation. This translation is the subject of this paper. Other aspects of SDD-1 are presented in [BSR, HS, RBFG].

The SDD-1 system architecture is described in [RBFG]. For purposes of this paper a simplified model will suffice. The system consists of a collection of sites fully connected by a communication network. Each site is a full-scale computer (as opposed to a micro-computer) and manages a portion of the database using a local database management system (abbr. DBMS). The database and each local DBMS are assumed to be relational; a review of relational terminology appears in Figure 1.1. The network is logically a point-to-point network (i.e., it does not support point-to-multipoint broadcast), and is assumed to have Arpanet-like performance characteristics[1].

The critical query processing problem in this environment is one of query optimization. Sustainable bandwidth on Arpanet is at most 10,000 bits per second; this is some three orders of magnitude lower than transfer rates between disk and main memory in typical full-scale computers. As a consequence, processing strategies with good performance in a centralized DBMS can easily explode in a distributed environment, running hundreds of times more slowly. Our principle objective is to avoid this performance degradation.

Stating our query optimization problem more precisely, we are given a transaction T and a database D which is statically distributed without replication[2]; our goal is to compute T(D) with a minimum quantity of inter-site data transfer. That is, we assume network bandwidth to be the system bottleneck, and our optimization objective is to minimize use of this resource. Other resources, notably

[1] SDD-1 is implemented on Arpanet.
[2] SDD-1's nandling of replicated data is discussed in Section 5.

Review of the Relational Data Model

Figure 1.1

(a) Relational Data Objects

Term	Definition
domain	a set of values
attribute	an alternate name for a domain
relation schema	a description of a relation, consisting of a relation name and list of attributes
relation	a subset of the cartesian product of the domains of the attributes of the corresponding relation schema
tuple	an element (or row) of a relation
database	a set of relations

(b) Relational Algebraic Operations

Selection: $R[A=x] = \{r \in R \mid r.A=x\}$
where r.A is the value of the A-domain in tuple r

Projection: $R[A_1,A_2,\ldots,A_n] = \{\langle r.A_1,r.A_2,\ldots,r.A_n\rangle \mid r \in R\}$

Join: $R[A=B]S = \{rs \mid r \in R,\ s \in S,\ \text{and}\ r.A = s.B\}$

local DBMS computation, are assumed to be *free*; in practice, local DBMS activity would be optimized as a secondary objective, but this issue will not be considered here.

Other cost factors we ignore include distance effects, the effects of network loading, and the overhead costs incurred whenever sites interact. We believe the first two effects to have second-order importance only. The third factor has much greater importance and precludes query processing strategies that employ large numbers of interactions [RGM]. Although we do not consider this factor explicitly, it is taken into account by the structure of the processing strategies we consider. As the reader will see, we always translate transactions into programs with relatively few interactions.

Our solution has two main steps. The first step translates the user's Datalanguage transaction into an internal QUEL-like form [HSW]. All aspects of query processing that depend on Datalanguage are handled in this first step. The second step optimizes the processing of the internal form. This step is quite general and can be used without modification in other distributed database systems. This paper emphasizes the optimization techniques of step two which we consider to be our principal contribution.

The paper is organized in six sections. Section 2 develops our paradigm for transaction execution, and defines the internal form that we subsequently optimize. Sections 3 and 4 describe the optimization of this internal form: Section 3 defines the "solution space" of the optimization -- i.e., the types of operations available for processing the internal form; and Section 4 presents our optimization algorithm for producing efficient sequences of these operations. The mapping from Datalanguage to internal form is explained in Section 5. Section 6 summarizes our technique and suggests extensions.

An early version of the SDD-1 query processing algorithm is described in [Wong]. Other approaches to distributed query processing appear in [ESW, HY, Willcox].

2. Query Processing Paradigm

Perhaps the simplest strategy for processing a transaction T against a distributed database is to move all relations referenced by T to a single site, and then execute T at that site. The disadvantage of this strategy is that it incurs unacceptably high communication cost. Our query processing paradigm is to perturb this simple strategy into an efficient one by using relational operations to reduce the size of each relation before moving it.

Distributed query optimization in our paradigm is concerned with performing this "reduction" process correctly and efficiently.

2.1 Reduction

Database state $D'=\{R'_1,...,R'_n\}$ is a sub-state of $D=\{R_1,...,R_n\}$ if R'_i can be obtained from R_i by selection and projection operations, for i=1,...,n. A reduction of database state D relative to transaction T is any sub-state D' such that T(D')=T(D). Intuitively, a reduction eliminates portions of the database that are irrelevant to T. In general, many reductions exist for each T and D. Given T and D our optimization task is to compile T into a program RHO such that

a. RHO(D) is a reduction of D relative to T,

b. all relations in RHO(D) are present at a single site, and

c. RHO incurs minimum cost (when applied to D) over all programs satisfying (a) and (b).

If RHO satisfies (a) and (b) for all D, then RHO is called a reducer for T.

2.2 Envelopes

To construct the desired program RHO, we find it necessary to analyze the body of T. However, Datalanguage transactions are approximately as general as programs written in a high-level programming language and it is difficult to analyze them directly. Therefore, we map each Datalanguage transaction into a QUEL-like internal form called an envelope, and optimize the envelope instead of the transaction. The mapping from transaction to envelope is dependent, of course, on details of

Datalanguage, and hence is specific to SDD-1; this transformation is described in Section 5. Having obtained an envelope, however, the remainder of our technique is applicable to other distributed relational DBMSs.

Syntactically, an envelope is essentially a QUEL query. An envelope, $E_{q,t}$, consists of a qualification, q, and a target list, t. A qualification is a boolean combination of selection clauses of the form (R.A=constant) and join clauses of the form (R.A=S.B), where R.A and S.B are indexed-variables and denote attribute A of relation R and attribute B of relation S respectively[3]. We assume that qualifications are pure conjunctions; disjunction is handled by placing the qualification in disjunctive normal form and treating each conjunction separately. A target list is a set of indexed-variables.

The result of applying $E_{q,t}$ to database D is defined by the following procedure:

[3] Note that we avoid tuple variables. Tuple variables can be accommodated by (conceptually) duplicating a relation and thereby having two relation-names range over it. We also avoid more general clauses, e.g., R.A<S.B, for pedagogic simplicity. They can added without altering the technical claims that follow.

1. Solve $E_{q,t}(D)$ as a query, using QUEL semantics; i.e.

   a. construct the cartesian product of the relations in D;

   b. eliminate tuples from the cartesian product that fail to satisfy qualification q; and

   c. project the remaining cartesian product onto the target-list t.

2. For each relation, R, project the result of (1) onto the attributes of R referenced in t, thereby producing a sub-state of D.

E is an envelope for T if for all database states D, T(E(D))=T(D), i.e. E(D) is a reduction of D relative to T. Intuitively, an envelope for T "envelopes" or delimits the portions of the database needed to process T. In general there are many envelopes for a given transaction; a good envelope is one that tightly delimits the data needed by T. Finding good envelopes is an optimization problem that depends on the language for expressing transactions. The solution used by SDD-1 appears in Section 5, but a general solution is not attempted. Figures 2.1-2.3 illustrate a database, a Datalanguage transaction, and an envelope for it.

Example Database

Figure 2.1

Relation Schema	Location of the Relation
SUPPLIER(S#, NAME, STREET, CITY, STATE)	site 1
SUPPLY(S#, P#, QTY, PRICE)	site 2
PART(P#, FUNCTION, SPEED, PACKAGE)	site 3

Example Transaction T_1

Figure 2.2

Note: Datalanguage constructs used in this example are explained in Section 5.

Description of transaction:

For each 7401-equivalent part supplied by a Massachusetts supplier, print the supplier number, name, address, price, and part number. In addition, print how many of these parts have switching speeds of 2 nano-seconds.

Transaction T_1:

```
  Begin
    Count:=0;
    For SUPPLIER
      If SUPPLIER.STATE="MA"
      Then For SUPPLY
           If SUPPLIER.S#=SUPPLY.S#
           Then For PART
                If SUPPLY.P#=PART.P# and PART.FUNCTION=7401
                Then Begin
                     Print SUPPLIER.S#, SUPPLIER.NAME,
                           SUPPLIER.STREET, SUPPLIER.CITY,
                           SUPPLIER.STATE, SUPPLY.PRICE,
                           PART.P#;
                     If PART.SPEED=2
                     Then COUNT:=COUNT+1;
                     End
      Print "Number of 2-nanosecond versions is", COUNT;
  End
```

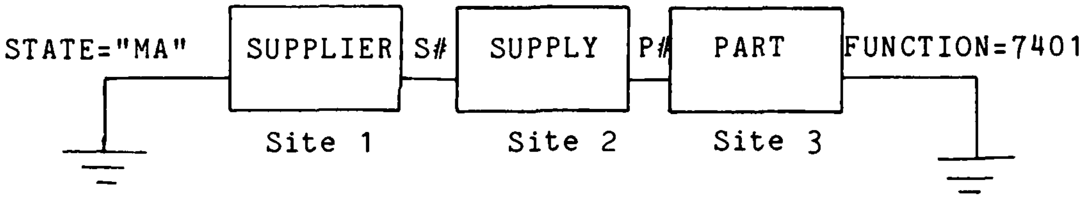
Example Envelope E_1, for T_1 Figure 2.3

Envelope E_1:

```
  target-list:{SUPPLIER.S#, SUPPLIER.NAME, SUPPLIER.STREET,
               SUPPLIER.CITY, SUPPLIER.STATE, SUPPLY.S#,
               SUPPLY.P#, SUPPLY.PRICE, PART.P#,
               PART.FUNCTION, PART.SPEED}

  qualification: SUPPLIER.STATE="MA"
             and SUPPLIER.S#=SUPPLY.S#
             and SUPPLY.P#=PART.P#
             and PART.FUNCTION=7401
```

graph representation of the envelope:

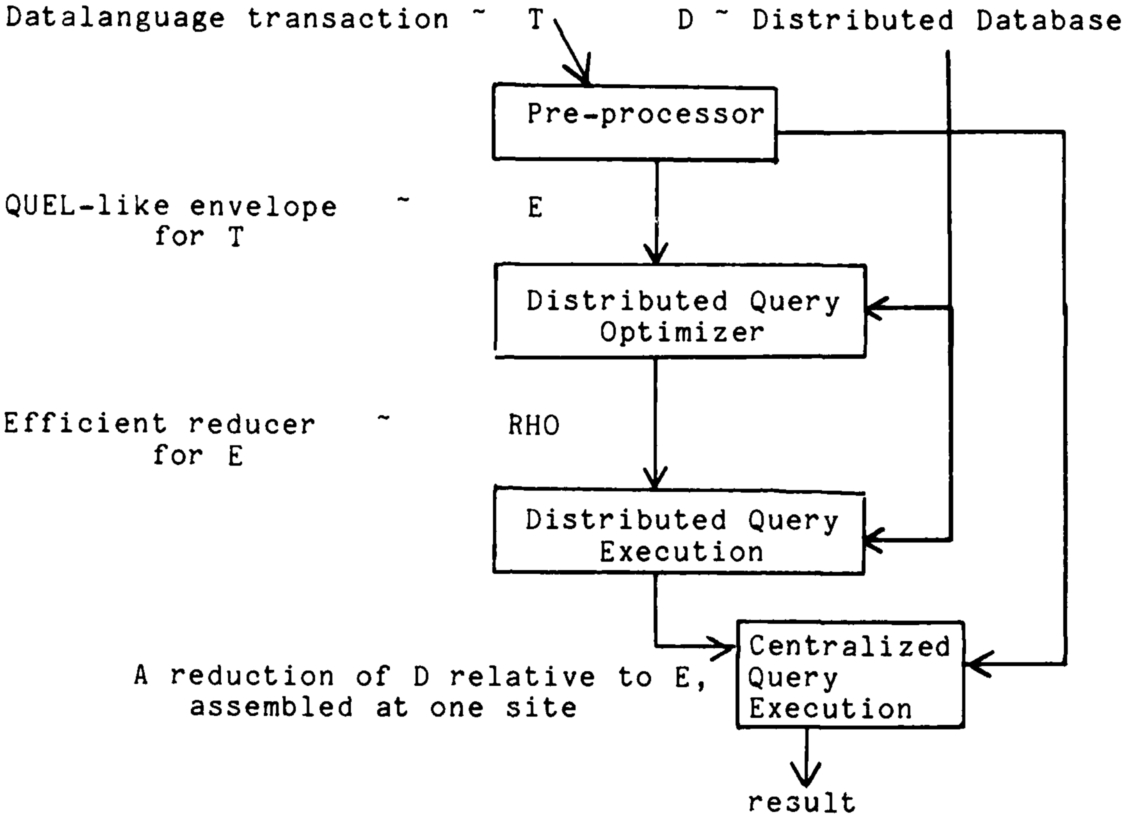


Importantly, if E is an envelope for T, then every reduction of a state D relative to E is also a reduction relative to T (i.e., E(D')=E(D) implies T(D')=T(D)). By way of proof, let D' be a reduction of D relative to E; so E(D')=E(D) by definition of reduction, and T(E(D'))=T(E(D)). Since T(E(D))=T(D) by definition of envelope, we have T(E(D'))=T(D) as desired. Consequently, every reducer for E is also a reducer for T.

Update transactions are also handled by this paradigm. Suppose U is an update transaction and E is an envelope for U (i.e. U(E(D))=U(D)). E is processed exactly as in the retrieval case: a reduction relative to E is assembled at one site, and U is executed on that reduction at that site. The result is a temporary file that lists all data items modified by U and their new values. These modifications are propagated to sites holding copies of those data items, and are installed using techniques described in [BSR].

Thus we have mapped the problem of finding efficient reducers for Datalanguage transactions into the problem of finding efficient reducers for envelopes. The next two sections address this latter problem. This paradigm is outlined in Figure 2.4.

Query Processing Paradigm Figure 2.4


Datalanguage transaction ~ T
D ~ Distributed Database
Pre-processor
QUEL-like envelope ~ E
for T
Distributed Query Optimizer
Efficient reducer ~ RHO
for E
Distributed Query Execution
A reduction of D relative to E,
assembled at one site
Centralized Query Execution
re3ult

3. Reduction Operations

We model a reducer RHO as a sequential program[4] containing reducing statements and assembly statements. Reducing statements apply relational operations to the database to compute the desired reduction; assembly statements move the resulting reduction to an assembly site where the original transaction is subsequently executed. This section describes the operations used by reducing statements.

3.1 Reduction Tactics

An operation is called legal for E if it maps any reduction relative to E into another such reduction. The purpose of this subsection is to characterize the set of legal operations for E, denoted OMEGA(E).

---

[4] RHO is, however, executed in a way that exploits parallel processing. See Section 3.3, and [RBFG].

3.1.1 Projections and Selections

The set of legal projections for E is

{R[X] ¦ R is a relation referenced by E, and X is the set of all attributes of R referenced in E}.

The set of legal selections for E includes

{R[A=k] ¦ R.A=k is a clause of E},

although additional legal selections may be implied by transitivity (see Section 3.1.2). The legality of both sets of operations is obvious.

For example, given envelope E_1 of Figure 2.3, the following operations are legal, and can be used to reduce the database:

1. SUPPLIER[STATE="MA"],
2. SUPPLY[S#,P#,PRICE],
3. PART[P#,FUNCTION,SPEED], and
4. PART[FUNCTION = 7401].

Projections and selections have zero cost under the assumptions of Section 1, since they require no inter-site data transfer. Consequently, every legal one should be included in every reducer.

3.1.2 Semi-Joins

Semi-join is a relational operation that exploits the join clauses of E for reduction purposes.

A semi-join is "half" of a join; the *semi-join* of relation R by relation S on attribute A, denoted $R<A=A]S$, is defined to be $(R[A=A]S)[ATT_R]$, where ATT_R denotes the attributes of R. Equivalently,

$$R<A=A]S = \{r \in R \mid (\exists s \in S)(r.A = s.A)\}.$$

Intuitively, $R<A=A]S$ eliminates every tuple of R that fails to join with *any* tuple of S.

Since $R<A=A]S = R<A=A](S[A]) = R[A=A](S[A])$, not all of S is needed to compute $R<A=A]S$; only $S[A]$ is required. Thus if R and S are stored at different sites, $R<A=A]S$ can be computed by transmitting $S[A]$ to R's site; it is *not* necessary to ship all of S.

Semi-joins have several important properties that make them valuable.

i. $R<A=A]S \subseteq R$

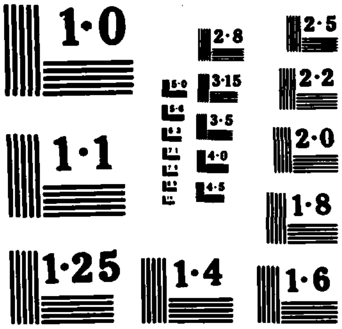
ii. $R[A=A]S = (R<A=A]S)[A=A]S$

AD-A155 724 A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND 4/4
AND CONTROL APPLICATIONS PART 2(U) COMPUTER CORP OF
AMERICA CAMBRIDGE MA 30 JAN 80 CCA-80-04-PT-2
UNCLASSIFIED N00039-77-C-0074 F/G 9/2 NL

END

FILMED

DTIC


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

iii. R[A=A]S = R[A=A](S<A=A]R) [5]

By (i), a semi-join can only decrease (never increase) the size of its left operand. By (ii) and (iii), a preliminary semi-join does not alter the result of a later join on the same clause. It follows that the set of legal semi-joins for E includes

{R<A=A]S, S<A=A]R ¦ R.A=S.A is a clause of E}.

As with selections, additional legal semi-joins may be implied by transitivity.

The set of operations implied by transitivity is obtained by constructing a node- and edge-labelled undirected graph G_E whose nodes are the indexed-variables and constants of E, and whose edges are

{{N_i,N_j} ¦ N_i=N_j is a clause of E}.

Then we construct the transitive closure of G_E, denoted G^{+}_{E}; G^{+}_{E} is a graph with the same nodes as G_E, but whose edges are

{{N_i,N_j} ¦ N_i and N_j are connected by a path in G_E}.

G^{+}_{E} can be computed efficiently using [Algorithm 5.2, AHU]. Given G^{+}_{E}, the set of legal selections for E is

{R[A=k] ¦ {R.A,k} is an edge of G^{+}_{E}},

[5] Unlike join, semi-join is not symmetric, hence R<A=A]S ≠ S<A=A]R. The former reduces R, while the latter reduces S.

and the set of legal semi-joins for E is

{R<A=A]S, S<A=A]R ¦ {R.A,S.A} is an edge of G_E^+}.

(Proofs appear in [BC,BG].)

Unlike selections, semi-joins generally have non-zero cost, and not all legal semi-joins are necessarily profitable. For example, the following semi-joins are both legal for envelope E_1:

1. SUPPLIER<S#=S#]SUPPLY, and
2. SUPPLY<P#=P#]PART,

If the database state has the characteristics shown in Figure 3.1, then the *cost* of the first semi-join equals the "size-of" SUPPLY[S#], which equals its width (i.e., the size of each tuple) multiplied by its cardinality (i.e., the number of tuples), which equals 1000. The *benefit* of the semi-join equals the amount by which it reduces SUPPLIER, which equals the size-of SUPPLIER minus the size-of SUPPLIER<S#=S#]SUPPLY; this benefit is at least 13*4000 = 52000, since at most 1000 SUPPLIER tuples can survive the semi-join. Thus, this semi-join is profitable. However, assuming SUPPLY[P#] $\subseteq$ PART[P#], the second semi-join is not profitable, since it does not reduce SUPPLY at all. Techniques for estimating costs and benefits of semi-joins are presented in Section 3.2.

Profile of Database from Figure 2.1 Figure 3.1

	SUPPLIER(	S#,	NAME,	STREET,	CITY,	STATE)
cardinality	5000	5000	-	-	-	50
width	13	1	3	3	3	3

	SUPPLY(	S#,	P#,	QTY,	PRICE)
cardinality	100000	1000	10000	-	-
width	4	1	1	1	1

	PART(	P#,	FUNCTION,	SPEED,	PACKAGE)
cardinality	10000	10000	200	-	
width	6	1	1	1	3

cardinality(domain(S#)) = 5000
cardinality(domain(P#)) = 10000

Legend:

cardinality = number of distinct values in a relation column, or an underlying domain.

width = number of bits, bytes, etc. per tuple, or column; widths are given in arbitrary units, with numeric fields having width 1 and string fields width 3.

blank entries are not relevant for the examples discussed in this paper.

Profiles are explained further in Section 3.2.

3.1.3 Join

Join is another operation that can potentially be used to reduce a database. We choose not to use join for this purpose, however, because the "reductive effect" of any single join can be obtained by using two semi-joins, usually at lower cost.

Let RS=R[A=A]S be an arbitrary join. Since our goal is to reduce the database, the relevant effect of this operation is its reductive effect on R and S; this effect is $RS[ATT_R]$ and $RS[ATT_S]$, where ATT_R and ATT_S denote the attributes of R and S respectively. Notice that

$$\begin{aligned} RS[ATT_S] &= \{s \mid \langle r,s\rangle \in RS\}, \text{ by definition a projection} \\ &= \{s \in S \mid (\exists r \in R)(r.A=s.A)\}, \\ &\qquad\qquad \text{by definition of } R[A=A]S \\ &= S\langle A=A]R, \text{ by definition of semi-join.} \end{aligned}$$

Thus the reductive effect of R[A=A]S on S can be attained by the semi-join S<A=A]R; by a similar argument, the effect on R can be attained by R<A=A]S.

Now let us compare the cost of the one join to that of the two semi-joins. To compute R[A=A]S, one of the relations, R say, must be shipped to the other's site.[6] Under the

[6] Techniques such as query feedback [Rothnie] may be able to decrease the quantity of data shipped, but intrcduce excessive inter-site interactions [RGM].

assumptions of Section 1, the cost of this operation equals the size-of R. To compute the semi-join, we ship R[A] to S and S[A] to R, for a cost of size-of R[A] + size-of S[A]. But S[A] $\bar{c}$ R[A] after S<A=A]R is executed. Thus if we execute the semi-joins in *sequence*, their cost is at most 2 * size-of R[A], which is at most size-of R under the (reasonable) assumption that the "width" of A is less than or equal to the "width" of ATT_R-{A}. Given this assumption, the cost of the semi-joins is less than or equal to the cost of the join as claimed.

If we consider *sequences* of joins, however, the preceding analysis is not always valid; there are cases in which the composite execution of multiple joins is more cost beneficial than the corresponding semi-joins [BC]. However, we believe these cases to be uncommon. Furthermore, such cases are difficult to detect from statistical database characteristics, such as those of Figure 3.1, because the difference in effect between joins and semi-joins depends in a detailed way on the database state [BC]. Therefore, we choose to ignore join as a distributed query processing tactic.

3.2 Performance Estimation

Hereafter we will be concerned with constructing a reducer RHO for E whose reducing statements are drawn from OMEGA(E). Since every omega $\in$ OMEGA(E) maps a reduction relative to E into another such reduction, every *sequence* of operations from OMEGA(E) also has this property; thus the logical correctness of RHO is guaranteed.

However, optimization considerations require that we construct a reducer that is *efficient* as well as correct. To do so we must estimate the performance of reduction operations. In particular, for each operation omega and database D, we need to estimate the *effect*, *cost*, and *benefit* of applying omega to D. Our techniques for this purpose are similar to those in [HY].

3.2.1 Profiles

To support these requirements, SDD-1 maintains a statistical description of the database, called a *profile*. Profiles contain the following information: For each relation R,

1. the number of tuples in R, denoted card(R);
2. the "width" of R, e.g. the number of bytes per tuple, denoted width(R) (we assume fixed-size tuples); and
3. for each attribute $A \in ATT_R$, the number of distinct values in R[A], denoted card(R[A]).

For each attribute A,

1. width(A) (we assume that A has the same width in each relation in which it appears); and
2. the number of distinct values in A's underlying domain, denoted card(dom(A)).

In using profiles, we assume that data values in each column of each relation are uniformly distributed over the tuples of the relation. We also assume all columns to be independent.

Profiles are updated off-line on a periodic basis to reduce overhead. The inaccuracies introduced by this time lag are acceptable because of the overall approximate nature of the process.

3.2.2 Effect Estimation

Let omega be an operation and $\bar{D}$ be a profile. The function *effect*(omega,$\bar{D}$) estimates the effect of omega on the database described by $\bar{D}$; its value is a profile $\bar{D}'$ that describes the (estimated) new state of that database.

If omega is a *projection*, e.g. R[X], *effect*(omega,$\bar{D}$) transforms width(R) into width(X) = $\mathrm{SUM}_{A \in X}$width(A). In general, R[X] can also reduce the cardinality of R by collapsing previously distinct tuples into a single tuple. We do not attempt to estimate this effect except in two cases:

1. if X={A}, then card(R) is changed to card(R[A]);
2. if $\mathrm{PRODUCT}_{A \in X}$(card(R[A])) < card(R), then card(R) is changed to equal that product.

A *selection*, e.g. R[A=k], affects card(R), and card(R[A']) for all $A' \in ATT_R$. Due to the uniformity assumption, the fraction of R tuples that satisfy the selection is

$$\mathrm{alpha}_k = \begin{cases} 1 \;/\; \mathrm{card(R[A])}, & \text{if } k \in R[A] \\ 0 & \text{, otherwise} \end{cases}$$

and the expected cardinality of the result is alpha_k * card(R). In practice it is prudent to assume $k \in R[A]$; if $k \notin R[A]$, the selection (and indeed the entire envelope) has a null result.

With this assumption, *effect*$(R[A=k],\bar{D})$ transforms card(R) into card(R)/card(R[A]), and card(R[A]) into 1. The effect on card(R[A']) for $A' \neq A$ is more complex and will be discussed momentarily.

The effect of a *semi-join* can be modelled as a sequence of selections. The fraction of R tuples expected to satisfy R<A=A]S is given by

$$\text{SUM}_{k \in S[A]} \text{alpha}_k$$

$$= \text{SUM}_{k \in S[A]} \; (1/\text{card}(R[A])) \; * \; (\text{the probability of } k \in R[A]).$$

Since we assume that columns are independent, the above probability is simply the probability that an *arbitrary* $k \in \text{dom}(A)$ is also in R[A], which equals card(R[A])/card(dom(A)). Substituting into the above formulas yields

$$\text{SUM}_{k \in S[A]}(1/\text{card}(\text{dom}(A)))$$

$$= \text{card}(S[A]) \; / \; \text{card}(\text{dom}(A)).$$

Thus the estimated cardinality of the result is

$$\text{card}(R) \; * \; \text{card}(S[A]) \; / \; \text{card}(\text{dom}(A)).$$

The effect of R<A=A]S on card(R[A]) is estimated similarly to be card(R[A])*card(S[A])/card(dom(A)).

The effect of R<A=A]S, or equivalently R[A=k], on R[A'] for A'≠A is more complex. Given the independence assumption, we can analyze the effect as a hit ratio problem: we are given n=card(R) "objects", distributed uniformly over m=card(R[A']) "colors"; the question is, "How many colors are we expected to hit if we randomly select r of the objects?" where r is the expected cardinality of the resulting relation. The answer is given by [Yao]:

$$Y(m,n,r) = m * (1 - \mathrm{PRODUCT}_{i=1}^{r}[(nd-i+1) / (n-i+1)],$$

$$\text{where } d = 1 - 1/m.$$

In practice, it is reasonable to approximate Y by

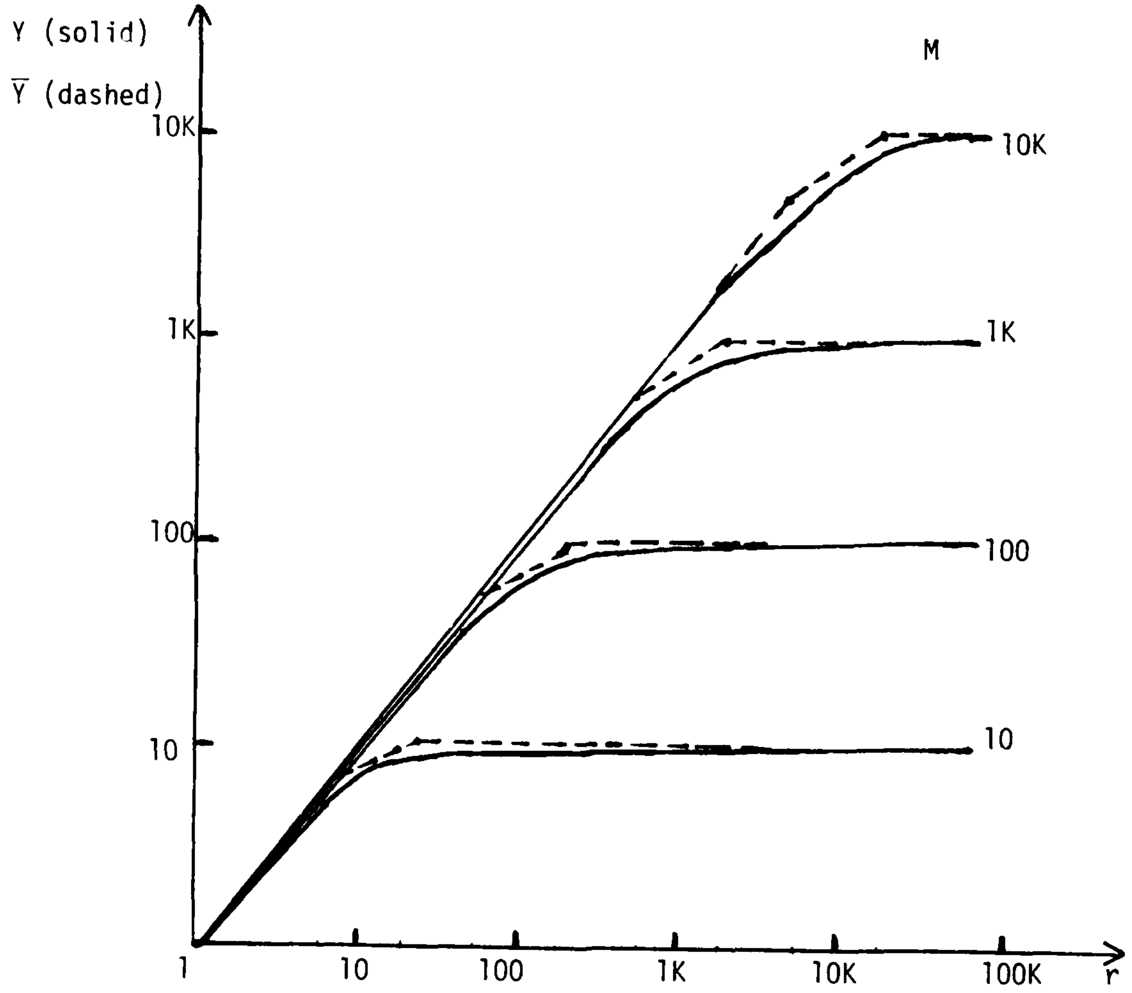
$$\bar{Y}(m,n,r) = \begin{cases} r & \text{, for } r<m/2 \\ (r+m)/3 & \text{, for } m/2 \leq r < 2m \\ m & \text{, for } 2m \leq r \end{cases}$$

Y and $\bar{Y}$ are graphed in Figure 3.2.

The Yao Function, Y, and Approximation, $\overline{Y}$. Figure 3.2

Hit ratio problem: given n objects distributed over m colors;
question: how many colors Y will we hit if we select r objects?

Fix n=100K



3.2.3 Cost Estimation

cost(omega,$\bar{D}$) is defined to be 0 for all local operations, i.e. projections, selections, and semi-joins whose operands are stored at a single site. If omega is a non-local semi-join R<A=A]S, then

cost(omega,$\bar{D}$)=card(S[A])*width(A).

3.2.4 Benefit Estimation

Suppose omega reduces relation R; its benefit is defined to be the amount by which it reduces R, which equals the size of R minus the size of omega(R). Substituting results from 3.2.2, we get

benefit(R[X],$\bar{D}$)=width(R)-width(X), assuming card(R) is not also changed;

benefit(R[A=k],$\bar{D}$)=width(R)*(card(R)-card(R)/card(R[A]))
=width(R)*card(R)*(1-1/card(R[A]));

benefit(R<A=A]S,$\bar{D}$)=
width(R)*card(R)*(1-card(S[A])/card(dom(A))).

3.3 An Example Reducer

To illustrate the preceding material we now present an example reducer for envelope E_1 of Figure 2.3. The initial database profile is given in Figure 3.1. The reducer proceeds as follows.

1. SUPPLIER[STATE="MA"]

   effect: card(SUPPLIER) reduced to 5000/50=100

   card(SUPPLIER[STATE]) reduced to 1

   card(SUPPLIER[S#]) reduced to Y(5000,5000,100)=100

   cost: 0

   benefit: 65000-1300=63700

2. SUPPLY[S#,P#,PRICE]

   effect: width(SUPPLY) reduced to 3

   cost: 0

   benefit: 100000

3. PART[P#,FUNCTION,SPEED]

   effect: width(PART) reduced to 3

   cost: 0

   benefit: 30000

4. PART[FUNCTION="7401"]

effect: card(PART) reduced to 10000/200=50

card(PART[FUNCTION]) reduced to 1

card(PART[P#]) reduced to Y(10000,10000,50)=50

cost: 0

benefit: 30000-150=29850.

5. SUPPLY<P#=P#]PART

effect: card(SUPPLY) reduced to 100000*50/10000=500

card(SUPPLY[P#]) reduced to 5000*50/10000=25

card(SUPPLY[S#]) reduced to
Y(1000,100000,500)=500

cost: card(PART[P#])*width(P#)=50

benefit: 300000-1500=298500

6. SUPPLY<S#=S#]SUPPLIER

effect: card(SUPPLY) reduced to 500*100/5000=10

card(SUPPLY[S#]) reduced to 500*100/5000=10

card(SUPPLY[P#]) reduced to Y(25,500,10)=10

cost: card(SUPPLIER[S#])*width(S#)=100

benefit: 1500-30=1470

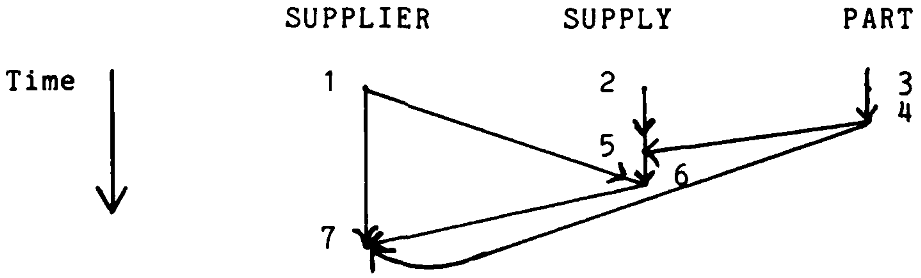
7. Assemble the reduction at site 1

cost: card(SUPPLY)*width(SUPPLY)
+ card(PART)*width(PART)=30+150=180.

The total cost of this reducer is 330. By comparison, if no reducing operations were performed at all, it would cost 125000 to assemble the database at one site (site 2 in this case). And if only local operations were performed -- i.e. steps 1-4 -- the cost would be 1450.

The flow-graph of this reducer is shown in Figure 3.3; nodes in this graph correspond to operations, and arcs indicate data-flow between operations. Each operation can be executed as soon as all of its predecessors in the flow-graph have been executed, and thus substantial parallellism is possible. This parallellism is exploited by *SDD-1 when it executes the reducer* [RBFG].

Flow-Graph of Example Reducer Figure 3.3



numbers relate nodes in the graph to operations in the text.

4. Access Planning

The development of Sections 2 and 3 have mapped the original query optimization problem into the following more structured problem: we are given an envelope E; our goal is to construct a reducer RHO for E whose reducing statements are drawn from OMEGA(E), and whose cost is minimum over all such reducers. We call this problem access planning. This section presents our access planning algorithm. We emphasize that our solution is approximate, seeking to find low-cost, though not necessarily optimal, reducers. An algorithm that produces optimal reducers for a limited class of envelopes is presented in [HY]; no efficient algorithm for producing optimal reducers for general envelopes is known.

4.1 Algorithm AP

Our access planning algorithm is Algorithm AP, listed in Figure 4.1. Algorithm AP in an iterative optimization procedure whose main function is to construct a profitable sequence of reducing statements. In general terms AP operates as follows. It initializes RHO to the null program and iteratively appends profitable, legal operations to RHO until all such operations have been used. Then the algorithm determines the cheapest site at which to assemble the reduction, and appends commands to move the reduction to that site. At this point RHO is the desired reducer, and the algorithm terminates.

We now examine AP in more detail.

Input/Output

The input to the algorithm is an envelope E and a database profile $\bar{D}$; its output is a reducer RHO for E whose reducing statements are estimated to be profitable when applied to the database described by $\bar{D}$.

State Space

The *state* of the algorithm at each iteration is determined by four variables.

1. RHO is the sequence of reducing statements constructed so far.
2. RHO($\bar{D}$) is the database profile that represents the estimated effect of applying RHO to the database described by $\bar{D}$; at each stage,

$$\mathrm{RHO}(\bar{D})=\underline{\mathrm{effect}}(\mathrm{omega}_k,(\underline{\mathrm{effect}}(\mathrm{omega}_{k-1},(\ldots,(\underline{\mathrm{effect}}(\mathrm{omega}_1,\bar{D}))\ldots)))),$$

   where $<\mathrm{omega}_1,\ldots,\mathrm{omega}_k>=\mathrm{RHO}$.

The Access Planner   Figure 4.1

Algorithm AP

Input: envelope E and database profile $\bar{D}$.
Output: RHO, a reducer for E.

State Space

RHO: a reducer for E.
RHO($\bar{D}$): database profile that results from executing RHO.
OMEGA: OMEGA(E)-{omega|omega is used in RHO}.
$OMEGA_{profitable}$: $\{omega \in OMEGA \mid benefit(omega,RHO(\bar{D})) \geq cost(omega,RHO(\bar{D}))\}$

Step 1 - Initialization

```
a.  RHO:=null program.
b.  RHO($\bar{D}$):=$\bar{D}$.
c.  OMEGA:=OMEGA(E).
d.  $OMEGA_{profitable}$:=$\{omega \in OMEGA(E) \mid benefit(omega,\bar{D}) \geq cost(omega,\bar{D})\}$
```

Step 2 - Main Loop

```
a. Do while $OMEGA_{profitable} \neq \emptyset$
b.    $omega_{best}$:=$omega \in OMEGA_{profitable}$ such that
        cost(omega,($\bar{D}$)) is minimum over all such omega;
        append $omega_{best}$ to RHO and remove from OMEGA and
        $OMEGA_{profitable}$.
c.      RHO($\bar{D}$):=effect($omega_{best}$,RHO($\bar{D}$));
        modify $OMEGA_{profitable}$ to reflect costs and
        benefits in new state (see text).
   end
```

Step 3 - Termination

```
a. select assembly site:
   - for each site s,
     $cost_a$(s)=SUM, over all relations R stored at s,
                  of width(R)*card(R);
   - the assembly site sa is the site s
     such that $cost_a$(s) is maximum over all sites.
b. append to RHO commands to move all relations to site sa.
```

END

3. OMEGA contains the legal operations not yet in RHO; these are the operations that can be added to RHO in future iterations. And

4. $\text{OMEGA}_{\text{profitable}}$ contains the operations that are estimated to be profitable in the state described by RHO($\bar{\text{D}}$).

Step 1 - Initialization

The four state variables are initialized to the appropriate values before the database has been reduced at all.

Step 2 - Main Loop

a. This step constructs a profitable sequence of reducing statements by repeating steps b & c until $\text{OMEGA}_{\text{profitable}}$ is exhausted.

b. On each iteration, the cheapest profitable operation, denoted $\text{omega}_{\text{best}}$, is appended to OMEGA.

An alternate approach would be to select the most profitable operation at each stage. However, suppose omega' has both high profit and high cost. Once we place omega' into RHO we have committed ourselves to paying its high cost. But if we delay omega' and execute other less costly operations first, these operations may reduce the cost of omega' as a fringe benefit.

Notice that all local operations have zero cost and non-negative benefit, hence are always profitable and always have lower cost than any non-local operation. Consequently all legal local operations are placed into RHO before any non-local operations are. In practice the order of these local operations is important; but since this order is a matter of local query optimization it does not concern us here.

This step also removes $\text{omega}_{\text{best}}$ from $\text{OMEGA}_{\text{profitable}}$. There are, however, cases in which it is beneficial to re-use the same operation, possibly many times [BC]. We choose to ignore this possibility because such cases apparently arise infrequently, and it is difficult to bound the size of RHO otherwise.

c. $\text{RHO}(\bar{D})$ is updated to reflect the estimated effect of $\text{omega}_{\text{best}}$, and $\text{OMEGA}_{\text{profitable}}$ is re-computed. To re-compute $\text{OMEGA}_{\text{profitable}}$, we need only check operations whose benefit or cost was changed by $\text{omega}_{\text{best}}$. In particular, suppose $\text{omega}_{\text{best}}$ is of the form R[X], R[A=k], or R<A=A]S. Then $\text{omega}_{\text{best}}$ reduces the size of R, and there are two further consequences:

1. the benefit of all other operations that reduce R is decreased; and
2. the cost of all semi-joins that use R to reduce another relation is also decreased.

Thus the operations that must be checked are:

1. $\{\text{omega} \in \text{OMEGA}_{\text{profitable}} | \text{omega reduces R}\}$, and
2. $\{\text{omega} \in \text{OMEGA} - \text{OMEGA}_{\text{profitable}}$
   $| \text{ omega} = S'<A'=A']R$, for any $S',A'\}$.

Step 3 - Termination

a. Upon termination of Step (2), RHO is a program that computes a reduction for E. To complete its task, RHO must also assemble the reduction at a single site.

   Let s1,...,sn be the sites housing data referenced by E, and let $\text{cost}_a(\text{si})$ be the sum, over all R at site si referenced by E, of width (R)*card(R). For any site sj, the cost of assembling the reduction at sj is

   $$\text{COST}_a(\text{sj})=\text{SUM}^{n}_{i=1,i\neq j}\ \text{cost}_a(\text{si}).$$

   COST_a is minimized by selecting the site with maximum cost_a to be the assembly site.

b. Having selected the assembly site, the algorithm appends commands to RHO to move all relations to that site. At this point, RHO is a reducer for E, and Algorithm AP terminates.

4.2 Enhancements

Algorithm AP is an example of a greedy optimization algorithm; it always makes decisions on the basis of immediate gain, it never looks ahead, and it never backs up. As a result, the reducers generated by AP are in general sub-optimal. In this section we present two techniques for improving these reducers. Both techniques take a reducer RHO produced by the basic algorithm and transform it into a lower cost reducer RHO'.

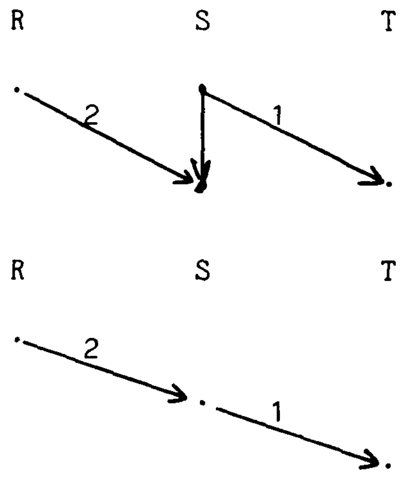
The first enhancement operates by permuting the order of RHO to reduce the cost of some semi-joins without increasing the cost of any other operations. This technique is best understood in terms of flow graphs. Consider Figure 4.2a. Since the semi-join represented by arc (2) reduces S, the cost of semi-join (1) can be decreased by delaying it until after (2). The resulting reducer RHO' is shown in Figure 4.2b. Notice that the reductive effect of semi-join (1) in RHO' is greater than its effect in RHO, because S[B] will be smaller in RHO', hence T<B=B]S will also be smaller. Consequently the cost of all semi-joins that follow (1) in the flow-graph are also reduced. Since no other semi-joins are affected by the transformation, RHO' is <u>guaranteed</u> to have lower cost than RHO.

More generally, let (N_R, N_S) be any arc in RHO's flow graph going from the "R column" to the "S column" and let N'_R be any node in the R column after N_R. The replacement of (N_R, N_S) by (N'_R, N_S) is guaranteed to monotonically decrease the cost of RHO, provided the resulting flow graph is acyclic. (If the resulting flow-graph contains a cycle, it no longer represents a physically executable program.)

To perform this transformation, we retain the values of RHO($\bar{D}$) computed at each step of the basic algorithm. When that algorithm terminates, we construct the flow graph of RHO and associate the retained values of RHO($\bar{D}$) with the corresponding nodes of the graph. Then we successively transform RHO by selecting the most expensive semi-join RHO and delaying it if possible: i.e. suppose (N_R, N_S) represents the most expensive semi-join in RHO, and let N'_R be the immediate successor of N_R in the R column, assuming N_R is not the last node in that column; we transform RHO

Improving RHO by Permuting Its Order Figure 4.2

qualification: R.A=S.A and S.B=T.B



(a) orginal reducer RHO

(b) better reducer RHO'

by replacing (N_R,N_S) by (N'_R,N_S) provided the resulting graph is acyclic. Values of RHO($\bar{D}$) associated with N_S and iLs successors in the graph are updated to reflect the transformation and the process repeats until no more transformations are possible.

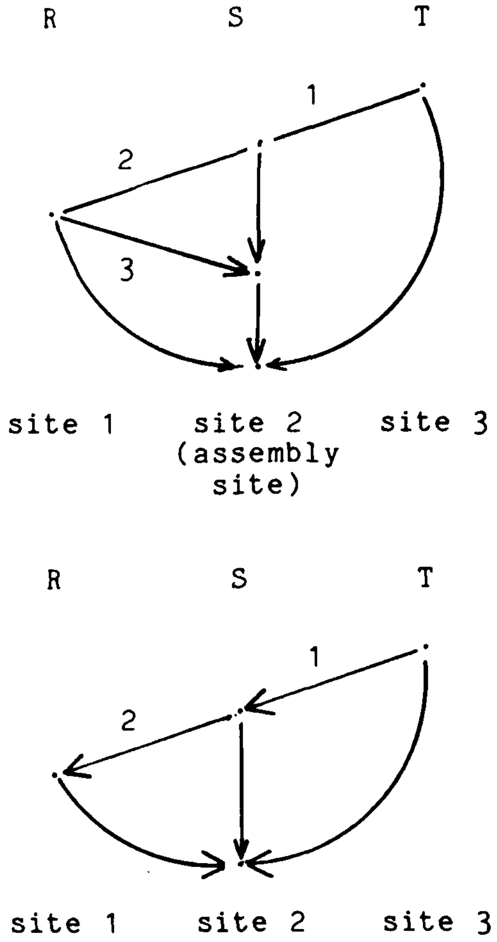
Our second enhancement seeks to prune operations from RHO that are rendered unprofitable by the choice of assembly site. Consider the reducer illustrated in Figure 4.3a, and suppose site 2 is the assembly site. Since relation S is stored at the assembly site, the semi-join S<A=A]R represented by arc (3) is superfluous and should be removed. The decision to incorporate this semi-join into

Improving by Pruning Semi-Joins — Figure 4.3

qualification: R.A=S.A and S.B=T.B

(a) original reducer RHO

(b) transformed reducer RHO'



RHO was based on the belief that S would eventually be shipped to the assembly site; since S is already there, the benefit of the semi-join is zero. With respect to arc (1), i.e. S<A=A]T, the situation is less clear-cut: although there is no direct benefit in reducing S, this

semi-join is indirectly beneficial via arc (2); in fact, arc (1) both decreases the cost of arc (2), and *increases its benefit*!

In general, let (N_R, N_S) be any arc in the flow graph for RHO where S is a relation stored at the assembly site. The removal of (N_R, N_S) is *beneficial* if the cost of RHO minus (N_R, N_S) is less than the cost of RHO (including all assembly operations); the former cost is computed by removing (N_R, N_S) from the strategy graph and updating values of RHO($\bar{D}$) associated with N_S and its successors. We perform this test on all arcs of the form (N_R, N_S), considered in cost order.

The enhancements described in this section help compensate for the short-sightedness of Algorithm AP, by considering the indirect benefits of semi-joins. While these enhancements still fall short of optimality, they move in that direction.

5. Mapping Transactions Into Envelopes

Sections 3 and 4 have described a technique for efficient processing of envelopes. In this section we explain the transformation from Datalanguage transactions to envelopes. This transformation is described in two steps. Section 5.1 describes the mapping from transactions to *logical* envelopes, i.e. envelopes that reference *logical* relations; Section 5.2 then describes the mapping to envelopes that reference *physical* relations.

5.1 The Logical Transformation

The purpose of the *logical transformation* is to eliminate procedural aspects of Datalanguage transactions. Datalanguage is a rich language and a full treatment of this mapping is beyond the scope of this paper. Instead, we will describe the mapping for a representative subset of the language.

5.1.1 A Subset of Datalanguage

A Datalanguage *transaction* is either a *statement* or a sequence of *statements* bracketed by Begin ... End. We will consider four types of *statements*: *iteration-statements*, *conditional-statements*, *assignment-statements*, and *print-statements*; Datalanguage does not include *goto*.

Two types of *variables* exist in Datalanguage: *indexed-variables* which represent database values (see below), and *program-variables* which are global variables in the style, say, of FORTRAN. Referring to transaction T_2 of Figure 5.1, PART.P# and PART.SPEED are *indexed-variables*, while COUNT is a *program-variable*.

Assignment and *print-statements* are self-explanatory and will not be described in detail. *Conditional-statements* have the form

If *boolean* then *body1* [else *body2*];

where *body1* and *body2* are either *statements* or sequences of *statements* bracketed by Begin ... End. *Conditional-statements* are interpreted with the usual semantics.

Transaction T_2 Figure 5.1

```
Begin
  COUNT:=0;
  For PART
     If PART.SPEED=2
     Then Begin
          If COUNT < 50
          Then Begin
            Print PART.P#;
            COUNT:=COUNT+1;
          End;;;
End
```

Iteration-statements have the form

```
For relation body1;
```

where body1 is defined above. This statement is interpreted as follows. body1 is executed once per tuple of relation with each indexed-variable of the form relation.attribute instantiated by the value of attribute in that tuple. For example, the statement

```
For PARTS
   If PARTS.FUNCTION=7401
   Then Print "Part Number =",PARTS.P#;;
```

prints the P# of every tuple in PARTS with FUNCTON=7401. We assume that each iteration-statement in a transaction references a different relation; if two iteration-statements reference the same relation a construct similar to "tuple variables" is employed.

5.1.2 Transformation to Logical Envelope

Given a transaction T, we obtain the target-list of its envelope by listing all *indexed-variables* that appear in T.

To obtain the qualification for T's envelope, the following recursive definition is applied. Let $\underline{\text{body}}=<\underline{\text{statement}}_1,\ldots,\underline{\text{statement}}_n>$. We define

$$\text{Qual}(\underline{\text{body}}) = \text{OR}_{i=1}^{n}\ \text{Qual}(\underline{\text{statement}}_i)\text{, where}$$

$$\text{Qual }(\underline{\text{statement}}_i) = \begin{cases} \text{(a) True, for } \underline{\text{assignment}} \text{ and} \\ \qquad\qquad \underline{\text{print statements}}; \\ \text{(b) } (\underline{\text{boolean}} \text{ AND Qual}(\underline{\text{body1}})) \\ \quad [\text{OR (NOT } \underline{\text{boolean}} \text{ AND Qual}(\underline{\text{body2}}))], \\ \quad \text{for } \underline{\text{conditional statements}}; \\ \text{(c) Qual}(\underline{\text{body1}}), \\ \qquad \text{for } \underline{\text{iteration statements}}. \end{cases}$$

The qualification for T's envelope is obtained by (1) removing all non-*iteration-statements* from T, yielding T', (2) constructing Qual(T'), and (3) replacing every clause that contains a *program-variable* by True. This procedure is illustrated in Figure 5.2.

It is easily proved that this transformation is correct, i.e. for all T it yields an envelope E such that T(E(D))=T(D) for all states of D. Moreover, if T contains no program-variables the transformation is exact, i.e. for all states of D, E(D) is the minimum state such that T(E(D))=T(D). Various code optimization techniques could be employed to improve the logical transformation when program-variables are present. However, this issue is not addressed here.

5.2 The Physical Transformation

At the physical level, SDD-1 permits each logical relation to be partitioned into sub-relations called fragments, which are the units of data distribution. Each fragment may be stored at one or more sites; each stored instance of a fragment is called a stored fragment. The relationship between relations, fragments, and stored fragments is illustrated in Figure 5.3. The purpose of the physical transformation is to map an envelope E that references logical relations into an envelope E_{SF} that references stored fragements.

The fragments of a relation are defined in two steps. First, the relation is partitioned "horizontally" into

Logical Transformation Figure 5.2

(1) Remove non-iteration-statements

```
   T'2:  < For PART
                   If PART.SPEED = 2
                   Then If COUNT < 50
                   Then Begin
                        Print PART.P#
                        COUNT:=COUNT+1;
                        END;;;>
```

(2) Construct Qual(T'_2)

= Qual(For PART ...;)

= Qual(If PART.SPEED = 2 Then ...;)

= PART.SPEED = 2 AND Qual(If COUNT < 50 Then ...;)

= PART.SPEED = 2 AND COUNT < 50 AND Qual(Print PART.P#; COUNT := COUNT + 1;)

= PART.SPEED = 2 AND COUNT < 50 AND (True OR True)

(3) Replace clauses that contain program-variables by True.

In this case, replace COUNT < 50 by True

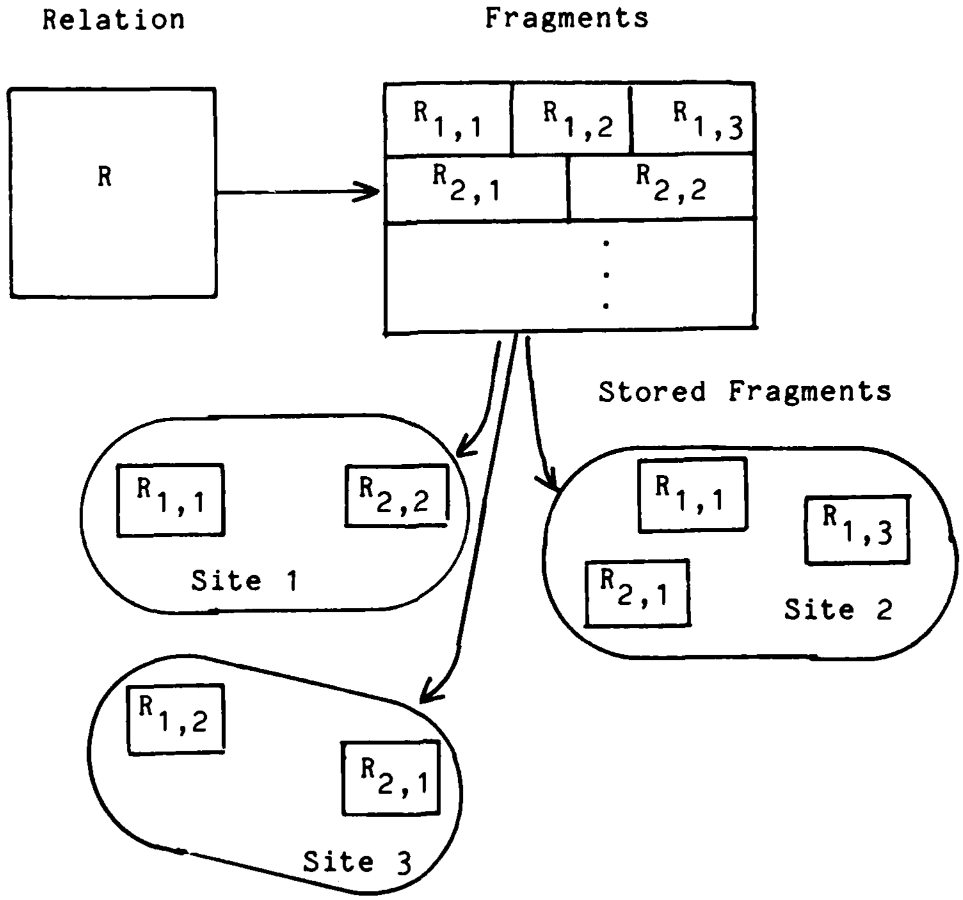
The resulting qualification is

PART.SPEED = 2

subsets defined by selection formulas (see Figure 5.4), and then each horizontal subset is partitioned vertically into sub-relations defined by projections (see Figure 5.5). In addition, a unique tuple identifier (abbr. TID) is appended to each tuple and included in every fragment

Relationship Between Relations and Stored Fragments

Figure 5.3



to guarantee lossless reconstruction of the original relation [DB]. We use $R_1,\ldots,R_n$ to denote the horizontal subsets of relation R, and $R_{i,1},\ldots,R_{i,m}$ to denote the vertical subrelations of R_i; notice that for all states of the database

$$R_i = R_{i,1}[\mathrm{TID=TID}]R_{i,2}[\mathrm{TID=TID}] \ldots [\mathrm{TID=TID}]R_{i,m}$$

and $R = R_1 \cup R_2 \cup \ldots \cup R_n$.

Horizontal Fragmentation Figure 5.4

Relation
PART(P#, FUNCTION, SPEED, PACKAGE)

Horizontal Fragments
PART_1=PART[SPEED=1]
PART_2=PART[SPEED=2 and PACKAGE="16 pin DIP"]
PART_3=PART[SPEED=2 and PACKAGE≠"16 pin DIP"]
PART_4=PART[SPEED>2]

Vertical Fragmentation Figure 5.5

1. Horizontal Fragment
   PART_1(P#, FUNCTION, SPEED, PACKAGE)

   Vertical Fragments
   $\text{PART}_{1,1}$=PART_1[P#, FUNCTION]
   $\text{PART}_{1,2}$=PART_1[SPEED]
   $\text{PART}_{1,3}$=PART_1[PACKAGE]

2. Horizontal Fragment
   PART_2(P#, FUNCTION, SPEED, PACKAGE)

   Vertical Fragments
   $\text{PART}_{2,1}$=PART_2[P#, SPEED]
   $\text{PART}_{2,2}$=PART_2[FUNCTION, PACKAGE]

3. etc.

Given an envelope E (referencing logical relations) we obtain an envelope E_F that references fragments by applying a query modification procedure described in [Dayal,DB]. This procedure maps each clause of the form R.A=S.A into a formula $OR^{nr}_{i=1}$ $(OR^{ns}_{j=1}$ $(R_i.A=S_j.A))$, where $R_1,\ldots,R_{nr}$ and $S_1,\ldots,S_{ns}$ are the horizontal fragments of

R and S respectively[7]. Then each indexed-variable $R_i.A$ is replaced by $R_{i,k}.A$, where $R_{i,k}$ is the vertical fragment of R_i that includes attribute A. (Since the vertical fragments partition R_i, the choice of $R_{i,k}$ is unique.) Finally, the vertical fragments of each horizontal fragment are "joined" on TID, by appending the formula $\mathrm{AND}_{k=1}^{mr-1}(R_{i,k}.TID=R_{i,k+1}.TID)$ to the qualification, where $R_{i,1},\ldots,R_{i,mr}$ are the vertical fragments of R_i referenced by the envelope. The result is the qualification of E_F; the target-list is obtained similarly.

E_F is then "improved" by detecting and discarding horizontal fragments whose definitions contradict the qualification. To do so, E_F is placed into disjunctive normal form and for each conjunct C the following test is performed. We append to C the selection formulas that define each horizontal fragment referenced in C. Then we test the satisfiability of the resulting formula using mechanical theorem-proving techniques. If the formula is unsatisfiable, C is removed from the qualification.

Given E_F, the remaining task is to obtain an envelope E_{SF} that references stored fragments. In principle, this

[7] Selection clauses of the form R.A=k are mapped into $\mathrm{OR}_{i=1}^{nr}\ R_i.A=k$.

transformation entails an optimization problem. It is, however, an optimization problem we choose not to address in SDD-1. Instead the mapping is accomplished via table look-up: for each site there is a pre-defined table, called a *materialization*, which specifies the stored instance of each fragment to use in processing transactions submitted at that site.

At this point, E_{SF} is an envelope in the form assumed by sections 2-4, and query processing proceeds as described there.

6. Conclusion

We have described query processing in SDD-1 as a three step procedure in which (1) a transaction is transformed into an envelope, (2) the envelope is compiled into a program called a reducer that assembles a reduction of the database at a single site, and (3) the transaction is executed against the reduction at that site. This approach separates issues that are transaction-language specific (steps (1) and 3)) from those of distributed query optimization (step (2)). This paper has concentrated on the latter issue; the optimization techniques presented in this paper are usable in other distributed relational DBMSs, as long as the translation from transactions to envelopes is feasible.

Our treatment has left many problems open. Among the most pressing are

1. finding ways of helping the optimization algorithm avoid entrapment by high-cost local optima;
2. use of feedback to compensate for inaccuracies in performance estimation; and

3. dynamic selection of stored fragments, e.g. to maximize the clustering of accessed fragments at individual sites.

We also believe that our methods can be extended to non-relational systems. Recent work by [Dayal, Zaniolo] on building relational interfaces to CODASYL databases suggest that our optimization techniques can be adapted to this setting. However, this too remains a matter for future research.

References

[AHU]

Aho, A.V., J. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[BC]

Bernstein, P.A., and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries," to appear JACM.

[BG]

Bernstein, P.A., and N. Goodman, "Full Reducers for Relational Queries Using Multi-Attribute Semi-Joins," Proc. 1979 NBS Symp. on Comp. Netw., Dec. 1979.

[BSR]

Bernstein, P.A., D.W. Shipman, and J.B. Rothnie, "Concurrency Control in SDD-1: A System for Distributed Databases", to appear ACM TODS.

[CCA]

Datacomputer Users Manual, Comp. Corp. of Am., Cambridge, MA, July, 1978.

[Dayal]

Dayal, U. Schema Mapping Problems in Database Systems, Tech. Rep. TR-11-79, Center for Research in Computing Technology, Harvard University, August, 1979.

[DB]

Dayal, U., and P.A. Bernstein, The Fragmentation Problem: Lossless Decomposition of Relations into Files, Tech. Rep. CCA-78-13, Comp. Corp. of Am., Cambridge, MA, November 1978.

[ESW]

Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing is a Relational Database System", Proc. 1978 Berkeley Work. on Dist. Data Management and Comp. Netw., May 1978

[HS]

Hammer, M.M., and D.W. Shipman, Reliability Mechanisms for SDD-1: A System for Distributed

Databases, Tech. Rep. CCA-79-05, Comp. Corp. of Am., Cambridge MA, July 1979.

[HSW]
Held, G., M. Stonebraker, and E. Wong, "INGRES: A Relational Database System", Proc. 1975 NCC, AFIPS Press, Montvale NJ.

[HY]
Hevner, A.R., and S.B. Yao, "Query Processing in Distributed Databases", IEEE Trans. on Soft. Eng., Vol SE-5, No. 3, May 1979.

[Rothnie]
Rothnie, J.B., "Evaluating Inter-Entry Retrieval Expressions in a Database Management System", Proc. 1975 NCC, AFIPS Press, Montvale NJ.

[RBFG]
Rothnie, J.B., P.A. Bernstein, S.A. Fox, N. Goodman, M.M. Hammer, T.A. Landers, D.W. Shipman, C.L. Reeve, and E. Wong, "SDD-1: A System for Distributed Databases", to appe ACM TODS.

[RGM]
Rothnie, J.B., N. Goodman, and T. Marill, "Database Architecture in a Network Environment", in Protocols and Techniques for Data Communication Networks, F.F. Kuo ed., Prentice-Hall, 1978.

[Willcox]
Willcox, D.A., "Optimization of a Relational Algebra Query to a Distributed Database Using Statistical Sampling Methods", Doc #234, Center for Advanced Computation, Univ. of Ill., August, 1977.

[Wong]
Wong, E., "Retrieving Dispersed Data in SDD-1: A System for Distributed Databases", Proc. 1977 Berkeley Workshop on Dist. Data Man. and Comp. Netw., May 1977.

[Yao]
Yao, S.B., "Approximating Block Accesses in Database Organizations", CACM, Vol. 20, No. 4, April 1977.

[Zaniolo]
Zaniolo, C., "Design of Relational Views Over

Network Schemes", Proc. 1979 ACM SIGMOD Conference, June 1979.